Utrecht University

Faculty of Science
Department of Information and
Computing Sciences

# Master's Thesis

# Chromatic $k$-Nearest Neighbors Queries using (Near-)Linear Space

*Author*
Thijs van der Horst
ICA-5873266

*Supervisor*
Dr. F. Staals
*Second examiner*
Dr. M. Löffler

June 27, 2021

**Abstract**

In this thesis, we study the *chromatic $k$-nearest neighbors* problem in one and two dimension(s), under the $L_1$, $L_2$ and $L_\infty$ metrics. In this problem, we wish to preprocess a set of $n$ colored points, such that given a query point $q$ and integer $k$, the most frequently occurring color among the $k$ nearest neighbors of $q$ can be found efficiently. We give the first data structures that answer queries without computing the $k$-nearest neighbors of a query point explicitly, resulting in query time complexities that are truly sublinear. Aside from the standard chromatic $k$-nearest neighbors problem, we also study the semi-online variant of the problem, in which we allow points to be added and deleted, as long as the time at which a point is deleted is known once it is inserted. Finally, we also consider an approximate version of the problem, in which a color needs to be reported that occurs often enough among the $k$-nearest neighbors of $q$. For all but the two-dimensional approximate problem under the $L_1$ and $L_\infty$ metrics, we give data structures that use linear space. For this last problem, we show the existence of a linear-space data structure, whose query time is lower than what we achieve.

# Contents

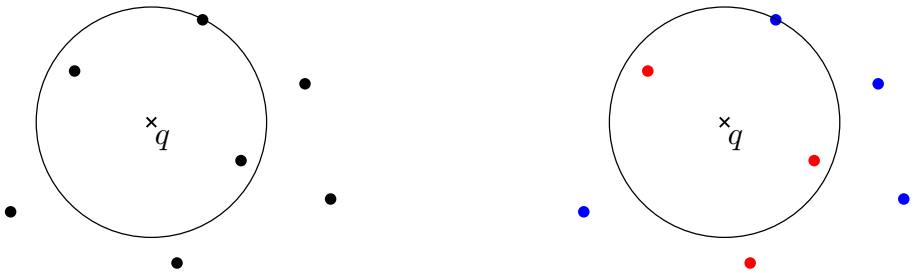



Figure 1: An illustration of the (chromatic) $k$-nearest neighbors problem. The points inside the circle on the left are the 3 nearest neighbors of $q$. On the right, the most frequently occurring color among the 3 nearest neighbors of $q$ is red.

# 1 Introduction

An important problem in applications dealing with e.g. pattern recognition and machine learning is that of *classification*. A known set of data points $P$ is partitioned into disjoint sets, called *classes*. Given a new data point $q$, not necessarily in $P$, we want to predict in which class $q$ belongs. Take for example the application of spam detection in an emailing service. We would use two classes, signifying "good" and "spam" email. A new email is then compared to a set of previously classified emails $P$, and is classified as either "good" or "spam" based on this set.

There is a wide variety of classification algorithms, or *classifiers*, available. One such classifier is the *$k$-nearest neighbors* classifier, which assigns a new point to the class occurring most frequently among the $k$ points in the data set that lie closest to this new point. This classifier is often used because of its performance and simplicity (see for example [11, 36, 40]).

The *chromatic $k$-nearest neighbors* problem, which is the topic of our thesis, is a geometric interpretation of the classifier. The problem is based on the *$k$-nearest neighbors* problem, in which we are given a set of $n$ points $P \subset \mathbb{R}^d$ to be preprocessed into a data structure, such that given a query point $q$, we can report the $k$ points from $P$ that lie closest to $q$. In the chromatic variant, every point in $P$ is given a color, and instead of reporting the $k$ points nearest to $q$, we have to report the color that occurs most frequently among these $k$ points. See Figure 1 for an illustration of the two problems. To show the connection with classification, the $k$-nearest neighbors classifier can be seen as classifying a data point based on the result of a chromatic $k$-nearest neighbors query.

## 1.1 Related work

The (non-chromatic) two-dimensional $k$-nearest neighbors problem has been studied extensively. In the case where $k$ is fixed during preprocessing, the problem can be solved by constructing an *order-$k$ Voronoi diagram*. This diagram is a generalization of the Voronoi diagram and was introduced by Shamos and Hoey [47]. While this diagram answers queries in optimal $O(\log n + k)$ time, the complexity of the diagram is $O(k(n-k))$ [15, 39, 42], and making it suitable for $k$-nearest neighbors queries requires $O(k^2(n-k))$ space.

When the Euclidean ($L_2$) metric is used, the problem of finding the $k$-nearest neighbors of a query point can be transformed to that of finding the $k$ planes in $\mathbb{R}^3$ whose intersections with a vertical query line are the lowest. This is accomplished by using a *lifting map* and working in dual space (see Section 5.1.1). In this setting, the $k$-nearest neighbors problem has been solved optimally by Afshani and Chan [2], who showed how to construct a data structure of $O(n)$ size in $O(n \log n)$ time, which answers queries in $O(\log n + k)$ time. Recently, Liu [41] studied the $k$-nearest neighbors problem under general metrics, including the $L_p$ metrics and additively weighted Euclidean metrics, and found a nearly optimal data structure that answers queries in optimal $O(\log n + k)$ time, but using $O(n \log \log n)$ space.

To answer queries efficiently in higher-dimensional space, it is common to use a *partition tree*, recursively partitioning the space into regions. An example of a tree storing a hierarchical subdivision is the $k$-d tree, introduced by Bentley [12]. When the points are randomly distributed, the tree supports $k$-nearest neighbors

queries in $O(\log n + k)$ expected time, regardless of dimension [33], making them theoretically optimal. In practice, however, the query times of these trees can deteriorate to $O(n)$ [51].

Another example is the balanced-box decomposition tree. Balanced-box decomposition trees have led to many different results regarding the $k$-nearest neighbors problem. The tree was introduced by Arya et al. [9], who obtained a result for an approximation variant of the $k$-nearest neighbors problem. In the approximate problem that they considered, a query needs to report $k$ points such that the $i$-th nearest reported point lies at most $(1 + \varepsilon)$ times as far from the query point as the $i$-th nearest neighbor of the query point. Their data structure achieves query times of $O_\varepsilon(k \log n)$, using $O(n)$ space (the $O_\varepsilon(\cdot)$ notation hides multiplicative factors depending only on $\varepsilon$). The trees have also been used by Mount et al. [45] to solve the exact chromatic problem, as well as the chromatic variant of the approximate problem studied by Arya et al. [9]. The data structure favors query points where there is a clear winner among the colors of the $k$ nearest neighbors. This uses the idea that in some applications, points are likely clustered together, and that points in a cluster will mostly be of the same color.

For the dynamic $k$-nearest neighbors problem, Agarwal et al. [3] gave a data structure of $O(n^{1+\delta})$[1] size, that can answer $k$-nearest neighbors queries under general distance functions in optimal $O(\log n + k)$ time, and which allows for insertions and deletions in $O(n^\delta)$ time. This was recently improved by Kaplan et al. [37], who found a dynamic data structure of $O(n \log^3 n)$ size that answers queries in $O(\log^2 n + k)$ time, which supports insertions and deletions in $O(\text{polylog}\, n)$ time. Soon after, Liu [41] improved this data structure, reducing its size to $O(n \log n)$.

## 1.2 Studied problems and results

In this thesis, we study the chromatic $k$-nearest neighbors problems in one and two dimension(s), under the $L_1$, $L_2$ and $L_\infty$ metrics. We make no assumptions for the parameter $k$ and the number of colors, other than that they are both at most equal to the number of points. Moreover, the parameter $k$ will not need to be known until query time, making our solutions support queries with varying values for $k$.

For these problems, we have to preprocess a set $P$ of $n$ points into a data structure, such that when given a query point $q$ and an integer $k \in [1, n]$, the most frequent color among the $k$-nearest neighbors of $q$ can be found efficiently. Note the dash in "$k$-nearest neighbors." We write this to signal that the $k$ points closest to $q$ do not have to be unique. The $k$-nearest neighbors of $q$ are defined as the points of $P$ for which at most $k - 1$ points lie strictly closer to $q$.

We assume that the points in $P$ lie in general position. This restriction can be worked around by symbolically perturbing the points [29, 31]. As the output size is constant, rather than $O(k)$ like for the regular $k$-nearest neighbors problem, we aim for query time complexities that depend only on $n$. See Table 1 for an overview of our results.

| Dimension | Metric | Preprocessing time | Space | Query time | Section |
|---|---|---|---|---|---|
| $d = 1$ | $L_1, L_2$ and $L_\infty$ | $O(n\sqrt{n})$ | $O(n)$ | $O(\sqrt{n})$ | 2 |
| $d = 2$ | $L_1$ and $L_\infty$ | $O(n^{5/3})$ | $O(n)$ | $O(n^{5/6})$ | 5 |
| | | $O(n^{5/3})$ | $O(n \log n)$ | $O(n^{2/3} \log n)$ | |
| | $L_2$ | $O(n^{5/3})$ | $O(n)$ | $O(n^{8/9})$ | |

Table 1: Our results for the exact chromatic $k$-nearest neighbors problems.

Aside from the static, exact problems, we give data structures for the dynamic (in one dimension) and *semi-online* (in two dimensions) versions of the problem. In the dynamic version of the problem, the set of points in the data structure can change over time, with points being added and deleted at unknown times. For the semi-online problem this is relaxed a bit, as the time at which a point will be deleted is given when the point is inserted (though the time at which a point is inserted remains unknown). See the work of Dobkin and Suri [26] for the introduction of semi-online problems, and the work of Chan [17] for more problems which were solved in a semi-online setting. Our results for the dynamic and semi-online chromatic $k$-nearest neighbors problems are given in Table 2.

[1] Throughout this thesis, we use $\delta$ to denote an arbitrarily small, positive constant.

| Dimension | Metric | Update time | Space | Query time | Section |
|---|---|---|---|---|---|
| $d = 1$ | $L_1, L_2$ and $L_\infty$ | $O(n^{2/3})$ | $O(n)$ | $O(n^{2/3})$ | 3 |
| $d = 2$ | $L_1$ and $L_\infty$ | $O(n^{6/7})^*$ | $O(n)$ | $O(n^{6/7} \log n)$ | 6 |
| | | $O(n^{3/4})^*$ | $O(n \log n)$ | $O(n^{3/4} \log^2 n)$ | |
| | $L_2$ | $O(n^{9/10})^*$ | $O(n)$ | $O(n^{9/10} \log n)$ | |

Table 2: Our results for the dynamic (in one dimension) and semi-online (in two dimensions) chromatic $k$-nearest neighbors problems. Complexities marked with $^*$ are amortized bounds.

Finally, we consider an approximate variant of the static problems. In the approximate version that we will consider, we are given a set $P$ of $n$ points, along with an approximation factor $\varepsilon \in (0, 1)$, which we have to preprocess into a data structure, such that given a query point $q$ and an integer $k \in [1, n]$, a color occurring at least $(1 - \varepsilon)$ times as often as the most frequent color among the $k$-nearest neighbors of $q$ can be found efficiently. See Table 3 for our results.

| Dimension | Metric | Preprocessing time | Space | Query time | Section |
|---|---|---|---|---|---|
| $d = 1$ | $L_1, L_2$ and $L_\infty$ | $O_\varepsilon(n \log n)$ | $O_\varepsilon(n)$ | $O_\varepsilon(\log n)$ | 4 |
| $d = 2$ | $L_1$ and $L_\infty$ | $O_\varepsilon(n^{1+\delta})^*$ | $O_\varepsilon(n \log^2 n)$ | $O_\varepsilon(n^{1/2+\delta})$ | 7 |
| | $L_2$ | $O_\varepsilon(n^{1+\delta})^*$ | $O_\varepsilon(n)$ | $O_\varepsilon(n^{1/2+\delta})$ | |

Table 3: Our results for the approximate chromatic $k$-nearest neighbors problems. The $O_\varepsilon(\cdot)$ notation hides multiplicative factors depending only on $\varepsilon$. Complexities marked with $^*$ are expected bounds.

# 2 The one-dimensional problem

In this section, we will solve the one-dimensional exact problem, which is the following.

ONE-DIMENSIONAL CHROMATIC $k$-NEAREST NEIGHBORS

**Given:** A set of $n$ colored points $P \subset \mathbb{R}$ in general position.

**Problem:** Preprocess $P$ into a data structure, such that given a query point $q \in \mathbb{R}$ and integer $k \in [1, n]$, the most frequent color among the $k$-nearest neighbors of $q$ can be found efficiently.

Section 2.1 shows how the problem can be reduced to the range mode problem in arrays. Section 2.2 then discusses a data structure by Chan et al. [20], which solves this range mode problem. Finally, in Section 2.3, we show how Chan et al. [20] achieved a conditional lower bound for the range mode problem, and show that this lower bound is also applicable to all our exact chromatic $k$-nearest neighbors problems.

Because points in $\mathbb{R}$ only have a single coordinate, we use points as values when we want to use the value of their coordinate. Also, because every $L_p$ metric is equivalent in $\mathbb{R}$, we will simply write $d(x, y) = |x - y|$ to mean the distance between two points $x, y \in \mathbb{R}$.

## 2.1 Finding the $k$-nearest neighbors

The main idea behind the data structure for finding the $k$-nearest neighbors is that while an order-$k$ Voronoi diagram in the plane has $O(k(n-k))$ regions [15,39,42], on the real line it consists of only $O(n)$ regions. We can therefore build an order-$k$ Voronoi diagram using $O(n)$ space, which we will use for finding the $k$-nearest neighbors of a query point. Furthermore, as we will see in this section, we do not need to build the diagram explicitly, which allows us to create the diagram for all values of $k$ simultaneously. This is necessary for our problem, where $k$ is not fixed.

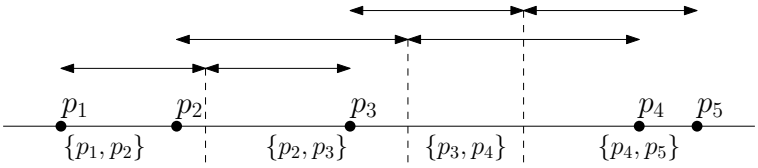



Figure 2: A one-dimensional order-$k$ Voronoi diagram, for $k = 2$. The dashed lines partition the real line into the order-$k$ Voronoi regions. The arrows above the dashed lines show that regions are bounded by bisectors of the points.

Let $p_1, \ldots, p_n$ be the points in $P$, ordered by increasing value. For a given $k$, we define the open interval $I_i$, which is analogous to a region in the order-$k$ Voronoi diagram, as

$$I_i = \begin{cases} \left(-\infty, \frac{p_i+p_{i+k}}{2}\right) & \text{if } i = 1 \\ \left(\frac{p_{i-1}+p_{i+k-1}}{2}, \frac{p_i+p_{i+k}}{2}\right) & \text{if } i \in \{2, \ldots, n-k\} \\ \left(\frac{p_{i-1}+p_{i+k-1}}{2}, \infty\right) & \text{if } i = n-k+1 \end{cases}.$$

Observe that for a point $q \in \mathbb{R}$, we either have that $q \in I_i$ for a unique $i \in \{1, \ldots, n-k+1\}$, or $q = \frac{p_i+p_{i+k}}{2}$ for some $i \in \{1, \ldots, n-k\}$. The points $\frac{p_i+p_{i+k}}{2}$ are analogous to edges in the order-$k$ Voronoi diagram, which are the bisectors between points. See Figure 2 for an illustration. Much like for regions and edges (and vertices) in an order-$k$ Voronoi diagram, points inside an interval $I_i$ have only a single $k$-th closest point, while a point $p = \frac{p_i+p_{i+k}}{2}$ has two that are equidistant from $p$. This leads to the following theorem.

**Theorem 2.1.** *Let $q \in \mathbb{R}$. The following statements hold:*

(i) *If $q \in I_i$ for some $i \in \{1, \ldots, n-k+1\}$, then $p_i, \ldots, p_{i+k-1}$ are the $k$-nearest neighbors of $q$.*

(ii) *If $q = \frac{p_i+p_{i+k}}{2}$ for some $i \in \{1, \ldots, n-k\}$, then $p_i, \ldots, p_{i+k}$ are the $k$-nearest neighbors of $q$.*

**Proof.**

(i) Assume that $q \in I_i$ for some $i \in \{1, \ldots, n-k+1\}$. Let $P_k = \{p_i, \ldots, p_{i+k-1}\}$. Split the points $P \setminus P_k$ up into two sets. Set $P_k^+ = \{p_{i+k}, \ldots, p_n\}$ contains the points to the right of $P_k$, and $P_k^- = \{p_1, \ldots, p_{i-1}\}$ contains the points to the left of $P_k$. Given $p \in P_k$ and $p^+ \in P_k^+$, we have that $q$ lies closer to $p$ than to $p^+$ if $q \in \left(-\infty, \frac{p+p^+}{2}\right)$. Therefore, we have that $d(q,p) < d(q,p^+)$ for all $p \in P_k$ and $p^+ \in P_k^+$ if

$$q \in \bigcap_{(p,p^+) \in P_k \times P_k^+} \left(-\infty, \frac{p+p^+}{2}\right) = \left(-\infty, \frac{p_i+p_{i+k}}{2}\right).$$

Similarly, given two points $p \in P_k$ and $p^- \in P_k^-$, we have that $q$ lies closer to $p$ than to $p^-$ if $q \in \left(\frac{p+p^-}{2}, \infty\right)$. Therefore, we have that $d(q,p) < d(q,p^-)$ for all $p \in P$ and $p^- \in P_k^-$ if

$$q \in \bigcap_{(p,p^-) \in P_k \times P_k^-} \left(\frac{p+p^-}{2}, \infty\right) = \left(\frac{p_{i+k-1}+p_{i-1}}{2}, \infty\right).$$

Note that $I_i$ is contained in both intervals, and therefore $q$ lies in both intervals. It follows that $P_k$ is the set of $|P_k|$-nearest neighbors of $q$. As $|P_k| = k$, this proves that $p_i, \ldots, p_{i+k-1}$ are the $k$-nearest neighbors of $q$.

(ii) Assume that $q = \frac{p_i+p_{i+k}}{2}$ for some $i \in \{1, \ldots, n-k\}$. Let $P_k = \{p_i, \ldots, p_{i+k}\}$. Let $P_k^+ = \{p_{i+k+1}, \ldots, p_n\}$ and $P_k^- = \{p_1, \ldots, p_{i-1}\}$. Given $p \in P_k$ and $p^+ \in P_k^+$, we have that $d(q,p) \leq d(q,p^+)$ if $q \in$

$\left(-\infty, \frac{p+p^+}{2}\right]$. Therefore, we have that $d(q,p) \leq d(q,p^+)$ for all $p \in P_k$ and $p^+ \in P_k^+$ if

$$q \in \bigcap_{(p,p^+) \in P_k \times P_k^+} \left(-\infty, \frac{p+p^+}{2}\right] = \left(-\infty, \frac{p_i + p_{i+k+1}}{2}\right].$$

Similarly, given $p \in P_k$ and $p^- \in P_k^-$, we have that $d(q,p) \leq d(q,p^-)$ if $q \in \left[\frac{p+p^-}{2}, \infty\right)$. Therefore, we have that $d(q,p) \leq d(q,p^-)$ for all $p \in P_k$ and $p^- \in P_k^-$ if

$$q \in \bigcap_{(p,p^-) \in P_k \times P_k^-} \left[\frac{p+p^-}{2}, \infty\right) = \left[\frac{p_{i+k} + p_{i-1}}{2}, \infty\right).$$

Note that $q$ lies in both intervals. It therefore follows that $P_k$ is the set of $|P_k|$-nearest neighbors of $q$, with $|P_k| = k+1 > k$. This means that the $k$-nearest neighbors of $q$ form a subset of $P_k$.

Because $q$ lies halfway between $p_i$ and $p_{i+k}$, it must be that $d(q,p) < d(q,p_i) = d(q,p_{i+k})$ for all $p \in \{p_{i+1}, \ldots, p_{i+k-1}\}$. The points $p_{i+1}, \ldots, p_{i+k-1}$ must therefore be part of the $k$-nearest neighbors. The only two points that can fill the last spot are $p_i$ and $p_{i+k}$, which both lie at the same distance from $q$. It follows that they both must be part of the $k$-nearest neighbors, proving that $p_i, \ldots, p_{i+k}$ are the $k$-nearest neighbors of $q$. □

**Corollary 2.2.** *Let $q \in \mathbb{R}$. There are two indices $1 \leq \ell < r \leq n$ such that $p_\ell, \ldots, p_r$ are the $k$-nearest neighbors of $q$.*

Let $p_\ell, \ldots, p_r$ be the $k$-nearest neighbors of the query point. Let $A$ be an array, storing the color of point $p_i$ at entry $A[i]$. The mode color among the $k$-nearest neighbors is then the mode color among the array entries $A[\ell : r]$. Therefore, once the indices $\ell$ and $r$ are found, this mode color can be determined by a range mode query.

We can find the indices $\ell$ and $r$ follows. First, check if $q > \frac{p_{n-k}+p_n}{2}$. If this is the case, then $q \in I_{n-k+1}$ and the $k$-nearest neighbors must be $p_{n-k+1}, \ldots, p_n$. This sets $\ell = n-k+1$ and $r = n$. Otherwise, we can find the highest value $i$ for which $q \leq \frac{p_i+p_{i+k}}{2}$ using binary search. Afterwards, we know that $p_i, \ldots, p_{i+k-1}$ are part of the $k$-nearest neighbors of $q$, and that $\ell = i$. A simple check to see if $q = \frac{p_i+p_{i+k}}{2}$ then tells us whether $p_{i+k}$ is also part of the $k$-nearest neighbors. If so, we have $r = i+k$. Otherwise, we have $r = i+k-1$. The result is an $O(\log n)$ time algorithm for reducing the problem to the range mode problem in an array.

## 2.2 Range mode queries in arrays

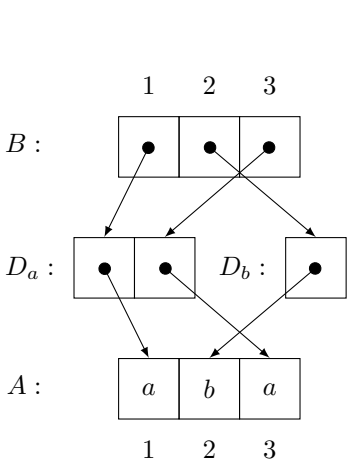
In [20], Chan et al. give multiple data structures for range mode queries in arrays. Their focus lies with the word-RAM model, but their first result is also applicable to our problem. We will discuss their data structure in this section.

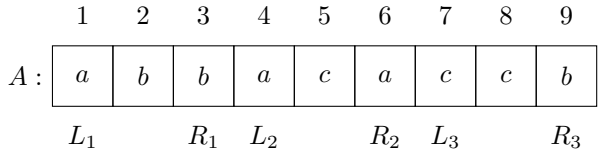
### 2.2.1 The data structure

Let $A[1:n]$ be an array of size $n$, storing colors. For ease of notation, we assume $n$ to be a perfect square. The data structure is easily altered to handle other cases. For the data structure, we divide $A$ into $\sqrt{n}$ blocks, each of size $\sqrt{n}$. We write $L_i = (i-1)\sqrt{n}+1$ to be the first index of the $i$-th block, and $R_i = i\sqrt{n}$ to be the last index of the $i$-th block, where $1 \leq i \leq \sqrt{n}$. The data structure then consists of the following parts (see Figure 3 for an illustration):

1. A table $S$, such that $S[i,j]$ stores the mode color of $A[L_i : R_j]$, along with its frequency in the range, where $1 \leq i \leq j \leq \sqrt{n}$. This table can be seen as storing the mode of all different sections of $A$ that are formed from contiguous blocks.

2. A sorted array $D_c$ for each color $c$, such that $D_c[i]$ stores the index in $A$ where the $i$-th occurrence of color $c$ is. These arrays serve as a quick lookup table to find the occurrences of a given color.




| i \ j | 1 | 2 | 3 |
|---|---|---|---|
| 1 | $(b, 2)$ | $(a, 3)$ | $(a, 3)$ |
| 2 | | $(a, 2)$ | $(c, 3)$ |
| 3 | | | $(c, 2)$ |

Figure 3: An illustration of the different data structures used. The figure on the left shows the relation between the different arrays. The figure on the right shows table $S$, computed for an array $A$.

3. An array $B[1 : n]$, such that $B[i]$ stores the index at which $i$ occurs in array $D_{A[i]}$. This array can be seen as storing pointers between the cells of $A$ and their corresponding cells in $D_{A[i]}$.

We now analyze the space and construction time complexities of the data structure. The table $S$ has size $O(n)$, as there are $\sqrt{n}$ blocks, and therefore $O(n)$ sets of contiguous blocks. We can compute all entries $S[i, j]$, for a fixed $i$, in a single scan of $A[L_i : n]$. To perform such a scan for a fixed $i$, set $j = i$. For each color $c$, keep track of the number of times $f_c$ that color $c$ has been encountered. Also keep track of the mode color $m$ and its frequency $f_m$ in the currently scanned part. When the frequency of a color $c$ is incremented to $f_c \leftarrow f_m + 1$, set $m \leftarrow c$ and $f_m \leftarrow f_c$. When we have reached index $R_j$ in the scan, we set $S[i, j] \leftarrow (m, f_m)$ and increment $j$, continuing the scan afterwards. As we need $\sqrt{n}$ scans to construct $S$, the table can be constructed in $O(n\sqrt{n})$ time.

Because every index $i$ is included in exactly one array $D_c$, all arrays $D_c$ combined take up $O(n)$ space. The arrays $D_c$ can all be constructed in a single scan of $A$, as the indices will be ordered already. They can therefore be built in $O(n)$ time. Finally, the array $B$ takes up $O(n)$ space, and can be built during the construction of the $D_c$ arrays in $O(n)$ time. The total space used is $O(n)$, and the total preprocessing time is $O(n\sqrt{n})$.

During a query, we want to quickly check whether color $A[\ell]$ occurs more than $f$ times in a range $A[\ell : r]$, for some $r \geq \ell$ and $f \geq 0$. This can be done in constant time with the above data structure, as shown in the following lemma.

**Lemma 2.3.** *Given the above data structure, we can determine in constant time whether $A[\ell : r]$ contains at least $f + 1$ instances of color $A[\ell]$, for any two indices $\ell$ and $r$ and for any integer $f \geq 0$.*

**Proof.** To check whether $A[\ell : r]$ contains at least $f + 1$ instances of $A[\ell]$, we merely have to check whether $D_{A[\ell]}[B[\ell] + f] \leq r$. This is because $D_{A[\ell]}$ stores the indices of all occurrences of element $A[\ell]$ in $A$. Index $B[\ell]$ of $D_{A[\ell]}$ therefore stores the first index $k$ between $\ell$ and $r$ for which $A[k] = A[\ell]$. This index will be $\ell$ itself. As $D_{A[\ell]}$ is a sorted array, it now follows that all elements $A[D_{A[\ell]}[k]]$ for $k$ between $B[\ell]$ and $B[\ell] + f$ are equal to $A[\ell]$ (assuming $D_{A[\ell]}$ contains enough entries). Therefore, if index $D_{A[\ell]}[B[\ell] + f]$ is at most $r$, we have that at least $f + 1$ colors in $A[\ell : r]$ are equal to $A[\ell]$, and otherwise this is not the case. As the check can be performed in constant time, the lemma is proven. □

### 2.2.2 The query algorithm

Given a query range $A[\ell : r]$, we first symbolically break up the query range into three parts. Let $L$ be the smallest index $L_i$ such that $L \geq \ell$, and let $R$ be the greatest index $R_i$ such that $R \leq r$. These two indices can be found using binary search in $O(\log n)$ time. We refer to range $A[L : R]$ as the *span* of the query range. The part left of the span, which is the range $A[\ell : \min\{L-1, r\}]$, will be referred to as the *prefix* of the range. Similarly, the part right of the span, which is the range $A[\max\{R+1, \ell\} : r]$ will be referred to as the *suffix* of the range. Note that these parts may be empty. Also, note that if the span is empty, the prefix and the suffix are not disjoint, and are both equal to the entire range. This will not break the query algorithm, and does not affect its complexity, though one can easily prevent this from happening by setting either of the two to empty.

The main part of the query algorithm is based on the following observation by Krizanc et al. [38].

**Observation 2.4.** Let $A$ and $B$ be multisets. If $c$ is a mode of $A \cup B$ and $c \notin A$, then $c$ is a mode of $B$.

Let $c = S[L, R]$ be the mode color of the span and let $f_c$ be its frequency in the span. Observation 2.4 now tells us that any element in the span, other than $c$, which does not occur in the prefix or suffix, will not be the mode of $B[\ell : r]$. Therefore, we now only have to consider $c$ and the elements in the prefix and suffix.

The frequency of the mode color in $A[\ell : r]$ must be at least $f_c$. Using Lemma 2.3, we can query whether a color $A[i]$ occurs at least $f_c + 1$ times in $A[i : r]$ constant time. As the prefix and suffix consist of at most $\sqrt{n}$ elements each, performing these queries on the prefix and suffix will take $O(\sqrt{n})$ time in total. A color in the prefix or suffix which occurs at least $f_c + 1$ times in the query range will be called a *candidate* color. The final part of the query algorithm consists of computing the frequencies in the query range of all candidate colors, and picking the one with the highest frequency.

We describe the procedure for computing the frequencies of colors in the prefix. The frequencies of the colors in the suffix can be computed analogously. Scan through the colors in the prefix from left to right. Let $i$ denote the index of the current item. The frequency of $A[i]$ will already have been counted if $A[i]$ occurs in $A[\ell : i-1]$. This can be checked by checking whether $D_{A[i]}[E[i]-1]$, which is the index of the previous occurrence of $A[i]$, is at least $\ell$. If so, color $A[i]$ occurs between indices $\ell$ and $i-1$, and thus will have been encountered already. If $A[i]$ has not been encountered yet, we check whether its frequency in $A[\ell : r]$ is at least $f_c + 1$ using Lemma 2.3. If it is, this color is a candidate color and we will compute its frequency in $A[\ell : r]$. Because $D_{A[i]}$ stores all indices of the occurrences of $A[i]$ in ascending order, we can compute the frequency of $A[i]$ through a linear scan of $D_{A[i]}$, starting at index $E[i] + f_c$ and ending at the first index $j$ for which $D_{A[i]}[j] > r$ (if the end of $D_{A[i]}$ is reached, we set $j = |D_{A[i]}| + 1$). The element $D_{A[i]}[j]$ denotes the first index in $A$ where color $A[i]$ lies outside the query range (if such an index exists). Note that we can start the scan at index $E[i] + f_c$ rather than index $E[i]$ because we know that the frequency of $A[i]$ is at least $f_c + 1$. The frequency of $A[i]$ in the query range is now $f_i = j - E[i]$. We then update the current mode color $c$ and its frequency $f_c$ with $A[i]$ and $f_i$, respectively, before continuing with the scan. After both the prefix and suffix have been scanned, the current mode color $c$ and its frequency $f_c$ are the mode color of $A[\ell : r]$ and its frequency, respectively.

What remains to be bounded for the query time is the time taken to perform all frequency counting. Note that the total number of elements inspected over the scans is $f_c - f'_c$, where $f'_c$ is the frequency stored in $S[L : R]$. This is because we start each scan at an offset equal to the frequency stored in $f_c$ before the scan, plus one. Because the mode color in $A[\ell : r]$ occurs at most $2\sqrt{n}$ times in the prefix and suffix, we have that $f_c$ and $f'_c$ differ by at most $2\sqrt{n}$. Therefore, the frequency counting, and thus a range mode query, takes $O(\sqrt{n})$ time.

**Theorem 2.5.** *Let $A$ be an array of size $n$. In $O(n\sqrt{n})$ time, we can build a data structure of $O(n)$ size, that answers range mode queries on $A$ in $O(\sqrt{n})$ time.*

Combining the result with that of Section 2.1, we obtain the following result for the chromatic $k$-nearest neighbors problem.

**Theorem 2.6.** *Let $P$ be a set of $n$ points in $\mathbb{R}$. In $O(n\sqrt{n})$ time, we can build a data structure of $O(n)$ size, that answers chromatic $k$-nearest neighbors queries on $P$ under the $L_1$, $L_2$ and $L_\infty$ metrics in $O(\sqrt{n})$ time.*

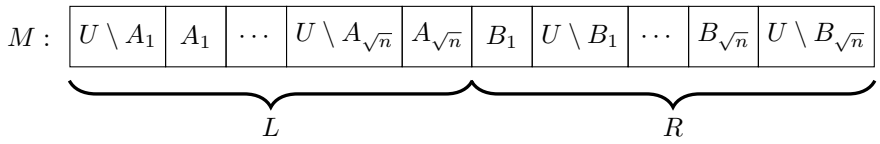



Figure 4: An illustration of the array $M$ constructed in the reduction from boolean matrix multiplication. The set $U$ contains the elements $1, \ldots, \sqrt{n}$.

## 2.3 A conditional lower bound

Aside from the $O(\sqrt{n})$ query-time data structure discussed in Section 2.2, Chan et al. [20] present strong evidence that a query time of $O(n^{1/2-\delta})$ cannot be obtained through combinatorial techniques. This is shown through a reduction from boolean matrix multiplication, which we will discuss here. As we will note at the end of this section, the resulting conditional lower bound also holds for our problem of chromatic $k$-nearest neighbors.

In the boolean matrix multiplication problem, we are given two square matrices $A$ and $B$, in which each entry is either 0 (False) or 1 (True). For the reduction, we assume that $\sqrt{n}$ is an integer, and that $A$ and $B$ are two $\sqrt{n} \times \sqrt{n}$ matrices. When multiplying the two matrices, we set an entry $c_{i,j}$ of the product matrix $C = A \cdot B$ as

$$c_{i,j} = \bigvee_{1 \le k \le \sqrt{n}} (a_{i,k} \wedge b_{k,j}) .$$

That is, multiplication of two entries is performed through the AND operator, and addition is performed using the OR operator. The reduction works by calculating each entry of $C$ with a separate range mode query, totalling $n$ queries.

For every row $i$ of $A$, we construct the set $A_i = \{k \mid a_{i,k} = 1\}$, containing those indices $k$ for which the entry is equal to 1. Similarly, we construct the set $B_j = \{k \mid b_{k,j} = 1\}$ for every row $j$ of $B$. An entry $c_{i,j}$ is now equal to 1 if and only if $A_i \cap B_j$ is non-empty. Whether this is the case can be tested through range mode queries, as observed by Greve et al. [34]:

**Observation 2.7** (Greve et al. [34])**.** Let $S_1$ and $S_2$ be sets (not multisets), and let $S$ be the multiset union of $S_1$ and $S_2$. The frequency of the mode of $S$ is one if $S_1 \cap S_2 = \emptyset$ and two otherwise.

To perform the $n$ required range mode queries, we first build an array $M$, which we divide into a left part $L$ and right part $R$. Part $L$ represents the rows of $A$, and consists of $\sqrt{n}$ blocks with $\sqrt{n}$ entries each. The $i$-th block, consisting of the entries $L[(i-1)\sqrt{n}+1 : i\sqrt{n}]$, represents row $i$ of $A$. The first entries are filled with the elements of $\{1, \ldots, \sqrt{n}\} \setminus A_i$, which are the indices $k$ for which entry $a_{i,k}$ is equal to 0. The order does not matter here. The last entries of the block are filled with the elements of $A_i$, again in some arbitrary order. The right part $R$ of $M$ is constructed similarly. It represents the columns of $B$, and consists of $\sqrt{n}$ blocks with $\sqrt{n}$ entries each. The entries of the $j$-th block are filled with the elements of $B_j$, in some arbitrary order, followed by the elements of $\{1, \ldots, \sqrt{n}\} \setminus B_j$, again in some arbitrary order. See Figure 4 for an illustration of the constructed array.

We can now determine whether $A_i \cap B_j$ is non-empty, and therefore whether $c_{i,j}$ equals 1, by performing a range mode query on the subarray $M[\text{start}(i) : \text{end}(j)]$, where

$$\text{start}(i) = (i-1)\sqrt{n} + 1 + \sqrt{n} - |A_i| \quad \text{and} \quad \text{end}(j) = n + (j-1)\sqrt{n} + |B_j|.$$

To see why, note that the elements at positions $\text{start}(i)$ through $\text{start}(i)+|A_i|-1$ are precisely the elements in $A_i$, and the elements at positions $\text{end}(j)-|B_j|+1$ through $\text{end}(j)$ are precisely the elements in $B_j$. Therefore, position $\text{start}(i) + |A_i|$ marks the start of a block of $M$, and position $\text{end}(j) - |B_j|$ mark the end of one. It follows that between $\text{start}(i)$ and $\text{end}(j)$ lie precisely the elements of $A_i$ and $B_j$, as well as $\sqrt{n} - i + j - 1$ blocks that each contain the elements of $\{1, \ldots, \sqrt{n}\}$ (see Figure 5). We now need another observation by Greve et al. [34].

**Observation 2.8.** Let $S$ be a multiset whose elements belong to some universe $U$. Adding one of each element in $U$ to $S$ increases the frequency of the mode of $S$ by one.

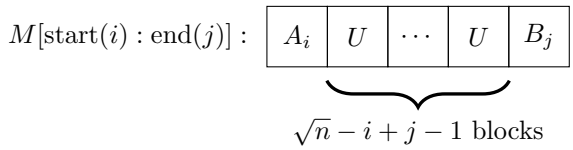



Figure 5: An illustration of the subarray $M[\text{start}(i) : \text{end}(j)]$ used in the reduction from boolean matrix multiplication. The set $U$ contains the elements $1, \ldots, \sqrt{n}$.

Recall that by Observation 2.7, $A_i \cap B_j$ is non-empty if and only if the frequency of the mode in the multiset union of $A_i$ and $B_j$ is two. The multiset of elements that lie between start($i$) and end($j$) is the multiset gotten from the union of $A_i$ and $B_j$, and adding the elements of $\{1, \ldots, \sqrt{n}\}$ a total of $\sqrt{n} - i + j - 1$ times. By Observation 2.8, it now follows that $A_i \cap B_j$ is non-empty if and only if the frequency of the mode in the subarray $M[\text{start}(i) : \text{end}(j)]$ is equal to $2 + \sqrt{n} - i + j - 1$.

To summarize, we can determine the value of an entry $c_{i,j}$ by performing a range mode query on the array $M$. Therefore, we can perform boolean matrix multiplication by first constructing $M$, and subsequently performing $n$ range mode queries on $M$. We can construct $M$, as well as find the cardinalities of the sets $A_i$ and $B_j$, in $O(n)$ time. This leads to the following result.

**Theorem 2.9.** *Assume that in $P(n)$ time, we can build a data structure that answers range mode queries in a given array in $Q(n)$ time. Then we can perform boolean matrix multiplication of two $\sqrt{n} \times \sqrt{n}$ matrices in $O(P(n) + n + n \cdot Q(n))$ time.*

While boolean matrix multiplication of two square matrices has been researched quite extensively (see for example [10, 19, 32, 53]), the best combinatorial bounds merely shave off logarithmic factors from the trivial time bound, which is $O(n^{3/2})$ for two $\sqrt{n} \times \sqrt{n}$ matrices. Combined with Theorem 2.9, this shows strong evidence that the query time of a range mode data structure for arrays must either have a worst-case preprocessing time that is $\omega(n^{3/2-\delta})$, or a worst-case query time that is $\omega(n^{1/2-\delta})$, assuming only combinatorial techniques may be used.

The lower bound is not only applicable to the range mode problem in arrays. In fact, it applies to all chromatic $k$-nearest neighbors problems. We show this through a reduction from range mode in arrays to one-dimensional chromatic $k$-nearest neighbors, which trivially extends to higher dimensions under all $L_p$ metrics.

The reduction is simple. Consider an array $A$. We map each entry $A[i]$ to the point $p_i = i$ in $\mathbb{R}$, and assign this point color $A[i]$. A query for the mode element inside subarray $A[\ell : r]$ can now be found as follows. Set $k = r - \ell + 1$. This is the number of elements in the query subarray. Let $q = \frac{\ell + r}{2}$ be the middle point among the points $p_\ell, \ldots, p_r$. Note that the $k$-nearest neighbors of $q$ are precisely the points $p_\ell, \ldots, p_r$. Therefore, the mode element of $A[\ell : r]$ is the mode color among the $k$-nearest neighbors of $q$. This results in the following theorem.

**Theorem 2.10.** *Assume that in $P(n)$ time, we can build a data structure that answers $d$-dimensional chromatic $k$-nearest neighbors queries under some $L_p$ metric in $Q(n)$ time, for some dimension $d$. Then we can perform boolean matrix multiplication of two $\sqrt{n} \times \sqrt{n}$ matrices in $O(P(n) + n + n \cdot Q(n))$ time.*

# 3 The one-dimensional dynamic problem

In this section we give a data structure for the one-dimensional dynamic problem, given as follows.

| Dynamic one-dimensional chromatic $k$-nearest neighbors | |
|---|---|
| **Problem:** | Build a data structure supporting insertions and deletions of points, such that given a query point $q \in \mathbb{R}$ and integer $k \in [1, n]$, the most frequent color among the $k$-nearest neighbors of $q$ can be found efficiently. |

The data structure from Section 2.1 is easily made dynamic. Recall that the data structure, when built on a set of points $P$, is merely an array storing the points in $P$ from left to right. A query looks for the left and

right indices $\ell$ and $r$, such that $p_\ell, \ldots, p_r$ are the $k$-nearest neighbors of a query point. To find these indices for a query point $q$, binary search was used over the integers $1, \ldots, n-k$ to find the highest integer $i$ such that $q \leq \frac{p_i + p_{i+k}}{2}$. The points $p_i, \ldots, p_{i+k-1}$ (and possibly $p_{i+k}$, if a check succeeds) are then the $k$-nearest neighbors of $q$.

Let $P$ be the $n$ points stored in the data structure at some point in time. Storing the points of $P$ from left to right, while allowing insertions and deletions, can be done using a red-black tree. This tree is a balanced binary-search tree that allows insertions and deletions in $O(\log n)$ time [24]. The query algorithm can be altered to use this tree, with a $O(\log n)$ factor overhead in the time complexity. This is because instead of being able to find a point $p_i$ in $O(1)$ time through a lookup in an array, we have to search through the red-black tree in $O(\log n)$ time. As the query algorithm for the static structure used $O(\log n)$ time, queries on the dynamic data structure take $O(\log^2 n)$ time.

Recall that by finding the indices $\ell$ and $r$ such that $p_\ell, \ldots, p_r$ are the $k$-nearest neighbors of a query point, the problem is reduced to the range mode problem in an array. We now show the dynamic data structure of El-Zein et al. [30] for the range mode problem in arrays, which directly results in a dynamic data structure for our one-dimensional chromatic $k$-nearest neighbors problem.

## 3.1 The data structure

Let $A[1:n]$ be an array of size $n$, storing colors, that the data structure is built on at some point in time. For ease of notation, we assume $n$ to be a perfect cube. The data structure is easily altered to handle other cases. The data structure is very similar to the static data structure of Chan et al. [20] that we discussed in Section 2.2. Like for the static data structure, we divide $A$ into blocks. This time however, we divide $A$ into $n^{1/3}$ blocks, each of size $n^{2/3}$. We write $L_i = (i-1) \cdot n^{2/3} + 1$ to be the first index of the $i$-th block, and $R_i = i \cdot n^{2/3}$ to be the last index of the $i$-th block, where $1 \leq i \leq n^{1/3}$. Aside from dividing $A$ into sections, we also divide the set of points into a set with *frequent* colors, which are colors that occur strictly more than $n^{1/3}$ times in $A$, and a set with *infrequent* colors, which occur at most $n^{1/3}$ times in $A$.

The data structure now consists of the following parts:

- A binary search tree $T$, storing the indices $R_i$. At each leaf storing index $R_i$, we store a red-black tree $T_i$, built on the frequent colors. This tree contains the colors and their frequencies in the range $A[0, R_i]$, sorted by frequency. The tree $T$ is a replacement for the table $S$ from Section 2.2.1. The reason that the table $S$ is not used anymore is because when inserting or deleting points, too many entries of $S$ would have to be changed.
- A *dynamic* sorted array $D_c$ for each color $c$, such that $D_c[i]$ stores the index in $A$ where the $i$-th occurrence of color $c$ is. These arrays are a dynamic variant of those from Section 2.2.1, and are given in [30].
- An array $B[1:n]$, such that $B[i]$ stores the index at which $i$ occurs in array $D_{A[i]}$. This array is the same as the one from Section 2.2.1.
- Arrays $F_{i,j}[0:n^{1/3}]$, for $1 \leq i \leq j \leq n^{1/3}$, such that $F_{i,j}[f]$ stores the number of infrequent colors with frequency $f$ in the range $A[L_i : R_j]$.

We now analyze the space and construction time complexities of the structure. There are $O(n^{2/3})$ arrays $F_{i,j}$, and each has length $n^{1/3}+1$. The space used by these arrays is therefore $O(n)$. A dynamic array built on $n$ indices uses $O(n)$ space [30]. Note that each index of $A$ is stored in exactly one dynamic array $D_c$. The total number of indices stored in the dynamic arrays is thus $O(n)$, and the space used by the arrays is $O(n)$ as well. Array $B$ trivially takes up $O(n)$ space. Finally, the tree $T$ has $O(n^{1/3})$ nodes. In each node, we store a binary search tree that is built on the frequent elements. These inner binary search trees therefore use $O(n^{2/3})$ space each. This amounts to $O(n)$ space in total for the tree $T$. Adding everything up, we see that the total space used by the data structure is $O(n)$.

To construct the data structure, we first split the elements into frequent and infrequent elements. This can be done by sorting $A$ and then performing a scan over $A$, in which we can count the frequencies of all elements. This takes $O(n \log n)$ time in total. We then construct the arrays $F_{i,j}$ by scanning $A$ a total of $n^{1/3}$ times, once for each $i$. To perform such a scan for a fixed $i$, set $j = i$. For each infrequent color $c$, keep track

of the number of times $f_c$ that color $c$ has been encountered. When an infrequent color $c$ is encountered, we first decrement $F_{i,j}[f_c]$. This is done as there is one less color with frequency $f_c$ now. We then increment its frequency $f_c \leftarrow f_c + 1$, and increment $F_{i,j}[f_c]$ to reflect this change. In total, we perform $O(1)$ operations for each element. When we have reached index $R_j$ in the scan, we "save" array $F_{i,j}$ and increment $j$, continuing the scan afterwards. This takes $O(n^{1/3})$ time per block, which is the time taken to copy array $F_{i,j}$. In total, we spend $O(n + n^{1/3} \cdot n^{1/3}) = O(n)$ time to construct the arrays $F_{i,j}$ for a fixed $i$, totalling $O(n^{4/3})$ time for all arrays $F_{i,j}$.

The dynamic arrays $D_c$ can be built through a single scan of $A$, as all indices will be ordered already. At element $A[i]$, index $i$ is inserted at the back of dynamic array $D_{A[i]}$. By [30], this operation takes $O(1)$ time, totalling $O(n)$ time. Finally, building $T$ can be done in one scan of $A$, by keeping track of the frequencies of the frequent elements encountered during the scan. Constructing a tree $T_i$ on the current frequencies takes $O(n^{2/3} \log n)$ time. As there are $O(n^{1/3})$ trees $T_i$ to build, the total time it takes to construct $T$ is $O(n \log n)$. The total construction time of the entire data structure is now $O(n^{4/3})$.

During a query, we want to quickly check whether color $A[\ell]$ occurs more than $f$ times in a range $A[\ell : r]$, for some $r \geq \ell$ and $f \geq 0$. This can be done in constant time with the above data structure, as shown in the following lemma.

**Lemma 3.1.** *Given the above data structure, we can determine in constant time whether $A[\ell : r]$ contains at least $f + 1$ instances of color $A[\ell]$, for any two indices $\ell$ and $r$ and for any integer $f \geq 0$.*

**Proof.** To check whether $A[\ell : r]$ contains at least $f + 1$ instances of $A[\ell]$, we merely have to check whether $D_{A[\ell]}[B[\ell] + f] \leq r$. This is because $D_{A[\ell]}$ stores the indices of all occurrences of element $A[\ell]$ in $A$. Index $B[\ell]$ of $D_{A[\ell]}$ therefore stores the first index $k$ between $\ell$ and $r$ for which $A[k] = A[\ell]$. This index will be $\ell$ itself. As $D_{A[\ell]}$ is a sorted array, it now follows that all elements $A[D_{A[\ell]}[k]]$ for $k$ between $B[\ell]$ and $B[\ell] + f$ are equal to $A[\ell]$ (assuming $D_{A[\ell]}$ contains enough entries). Therefore, if index $D_{A[\ell]}[B[\ell] + f]$ is at most $r$, we have that at least $f + 1$ colors in $A[\ell : r]$ are equal to $A[\ell]$, and otherwise this is not the case. As the check can be performed in constant time, the lemma is proven. □

## 3.2 The query algorithm

We now give the query algorithm. The algorithm works similarly to the query algorithm for the static data structure from Section 2.2. Given a query range $A[l : r]$, we first symbolically break up the query range into three parts. Let $L$ be the smallest index $L_i$ such that $L \geq \ell$, and let $R$ be the greatest index $R_i$ such that $R \leq r$. These two indices can be found using binary search in $O(\log n)$ time. Like in Section 2.2.2 refer to the range $A[L : R]$ as the *span* of the query range. The part left of the span, which is the range $A[\ell : \min\{L - 1, r\}]$, will be referred to as the *prefix* of the range. Similarly, the part right of the span, which is the range $A[\max\{\max\{\ell, R + 1\} : r]$ will be referred to as the *suffix* of the range. Note that these parts may be empty.

The main part of the query algorithm is again based on observation 2.4, which we restate here.

**Observation 3.2.** Let $A$ and $B$ be multisets. If $c$ is a mode of $A \cup B$ and $c \notin A$, then $c$ is a mode of $B$.

Rather than first finding the mode color of $A[L : R]$ using a precomputed table, we find the mode color of $A[L : R]$ *among the frequent colors*. To do this, we first locate the leaves in $T$ storing indices $L - 1$ and $R$. Then we scan the trees inside these leaves, to find the frequencies of all frequent elements in $A[0 : L - 1]$ and $A[0 : R]$. By subtracting the results from each other, we obtain all frequencies of the frequent elements in $A[L : R]$. Locating the correct leaves in $T$ takes $O(\log n)$ time, after which the scans over the two trees $T_{L-1}$ and $T_R$ take $O(n^{2/3})$ time. This step therefore takes $O(n^{2/3})$ time.

Let $c$ be the frequent color with the highest frequency in $A[L : R]$, and let $f_c$ be its frequency in $A[L : R]$. The frequency of the mode color in $A[\ell : r]$ must be at least $f_c$. Using Lemma 3.1, we can query whether a color $A[i]$ occurs at least $f_c + 1$ times in $A[i : r]$ in constant time. As the prefix and suffix consist of at most $n^{2/3}$ elements each, performing these queries on the prefix and suffix takes $O(n^{2/3})$ time in total. A color in the prefix or suffix which occurs at least $f_c + 1$ times in the query range will be called a *candidate* color. The next part of the query consists of computing the frequencies in the query range of all candidate colors.

We describe the procedure for computing the frequencies of colors in the prefix. The frequencies of the colors in the suffix can be computed analogously. Scan through the colors in the prefix from left to right. Let

$i$ denote the index of the current item. The frequency of $A[i]$ will already have been counted if $A[i]$ occurs in $A[\ell : i-1]$. This can be checked by checking whether $D_{A[i]}[B[i]-1]$, which is the index of the previous occurrence of $A[i]$, is at least $\ell$. If so, color $A[i]$ occurs between indices $\ell$ and $i-1$, and thus will have been encountered already. If $A[i]$ has not been encountered yet, we check whether its frequency in $A[\ell : r]$ is at least $f_c$ using Lemma 3.1. If it is, this color is a candidate color and we will compute its frequency in $A[\ell : r]$. Because $D_{A[i]}$ stores all indices of the occurrences of $A[i]$ in ascending order, we can compute the frequency of $A[i]$ through a linear scan of $D_{A[i]}$, starting at index $B[i]+f_c$ and ending at the first index $j$ for which $D_{A[i]}[j] > r$ (if the end of $D_{A[i]}$ is reached, we set $j = |D_{A[i]}|+1$). The element $D_{A[i]}[j]$ denotes the first index in $A$ where color $A[i]$ lies outside the query range (if such an index exists). Note that we can start the scan at index $B[i]+f_c$ rather than index $B[i]$ because we know that the frequency of $A[i]$ is at least $f_c+1$. Therefore, the frequency of $A[i]$ in the query range is $f_i = j - B[i]$. We then update the current mode color $c$ and its frequency $f_c$ with $A[i]$ and $f_i$, respectively, before continuing with the scan. After both the prefix and suffix have been scanned, the current mode color $c$ and its frequency $f_c$ are the mode color of $A[\ell : r]$ and its frequency, respectively.

We now bound the time taken to perform the frequency counting of all candidate colors. If $c$ is an infrequent color, we have that $f_c \leq n^{1/3}$. As for each scan, the value of $f$ must have increased by at least one, there are only $O(n^{1/3})$ scans performed, each taking $O(n^{1/3})$ time. The scans therefore take $O(n^{2/3})$ time in total when $c$ is an infrequent color. Now assume that $c$ is a frequent color. Note that the total number of elements inspected over the scans is $f_c - f'_c$, where $f'_c$ is the frequency of the mode in $A[L : R]$ among the frequent colors. This is because we start each scan at an offset equal to the frequency stored in $f_c$ before the scan, plus one. As $c$ is a frequent color, we have that $f'_c$ is at least the frequency of $c$ in $A[L : R]$. Because there are only $O(n^{2/3})$ elements in the prefix and span, we have that $f_c - f'_c$ is at most $O(n^{2/3})$. The total cost of the scans will therefore be $O(n^{2/3})$ in case $c$ is a frequent color, making the scans take $O(n^{2/3})$ time in any case.

Finally, by Observation 3.2, we have that the mode color in $A[\ell : r]$ is either $c$, or it is an element in the span which does not occur in either the prefix or the suffix. Furthermore, if it is an element of the span, it must also be the mode of $A[L : R]$. The final part of the query now deals with finding the mode of $A[L : R]$, and comparing its frequency in $A[L : R]$ to frequency $f_c$. Let $L_i = L$ and $R_j = R$. We first scan over $F_{i,j}$ in $O(n^{1/3})$ time to find the highest index $f_{i,j}$ with a non-zero entry. Frequency $f_{i,j}$ is the frequency of the mode color in the range $A[L : R]$. If $f_{i,j}$ is less than $f$, then we have that $c$ is the mode of $A[\ell : r]$. If $f_{i,j}$ is higher than $f$, then we must search for a color in $A[L : R]$ with this frequency. To do this, we first scan over all indices $R_k$ that lie left of $r$, from right to left. For every index $R_k$ encountered, we check how many colors have frequency $f_{i,j}$ in $A[L : R_k]$, by looking at $F_{i,k}[f_{i,j}]$. If this number is lower than $F_{i,j}[f_{i,j}]$, then there is a color in the block starting at index $R_k+1$ that has frequency $f_{i,j}$ in the range $A[L : R]$. In $O(n^{1/3})$ time, we can now find a block containing a mode color for $A[\ell : r]$. By scanning over this block, and using Lemma 3.1 with frequency $f_{i,j}$, we can find a mode color in $O(n^{2/3})$ extra time. The total query time is therefore $O(n^{2/3})$.

## 3.3 Updating elements in the data structure

Before we show how to insert and delete elements, we first show how to update an element to a different color. An update $A[i] \leftarrow c$ is handled by first removing the color $A[i]$ from the data structure, and then inserting color $c$ at index $i$. Note that this procedure is slightly different than inserting and deleting colors, as the insertion and deletion happen at the same index and right after each other.

Let $c$ be the color that is added or removed. If $c$ is infrequent before the update, we first count the frequency of $c$ in each section of contiguous blocks $A[L_i : R_j]$, for $1 \leq i \leq j \leq n^{1/3}$. We can compute these frequencies for a fixed $L_i$, and for every $R_j \geq L_i$, through a single scan of $D_c$. As $c$ is an infrequent color, the size of $D_c$ is at most $n^{1/3}$, so a single scan will take $O(n^{1/3})$ time. As there are $n^{1/3}$ indices $L_i$ for which we perform such a scan, the total time taken for the scans is $O(n^{2/3})$. Let $f_{i,j}$ be the frequency of $c$ in section $A[L_i : R_j]$. If the index at which the update happens lies between $L_i$ and $R_j$, we decrement entry $F_{i,j}[f_{i,j}]$. Doing this for all sections $A[L_i : R_j]$ takes $O(n^{2/3})$ time, making the total time taken for this step $O(n^{2/3})$.

If $c$ is infrequent after the update, we need to adjust the arrays $F_{i,j}$ again, as $c$ is now not included in them. To do this, let $f_{i,j}$ be the frequency of $c$ in section $A[L_i : R_j]$ *before the update*. We then either increment $F_{i,j}[f_{i,j}-1]$ (if $c$ was removed) or $F_{i,j}[f_{i,j}+1]$ (if $c$ was added). Doing this for all arrays $F_{i,j}$ takes

$O(n^{2/3})$ time in total, after which the arrays will be in a correct state.

The next step is to adjust the array $D_c$ (regardless of whether $c$ is infrequent or not). This is done by adding (in case $c$ is added) or removing (in case $c$ is removed) index $i$ from the array $D_c$. This can be done in $O(\sqrt{n})$ time [30]. If $c$ is added, we also need to adjust $B[i]$ to store the index in $D_c$ that stores index $i$. This can be done while adding $i$ to $D_c$, making this step take $O(\sqrt{n})$ time.

Finally, the tree $T$ needs to be adjusted. If $c$ was frequent before the update, but became infrequent after the update, we need to remove color $c$ (together with its frequencies) from all trees $T_i$ in $T$. As there are $O(n^{1/3})$ trees $T_i$ stored in $T$, and because the trees $T_i$ are red-black trees, we can remove color $c$ from all trees in $O(n^{1/3} \log n)$ time in total [24]. Similarly, if $c$ was infrequent but became frequent through the update, we need to add $c$ and its frequencies to the trees $T_i$. We can compute the frequencies of $c$ in $A[0, R_i]$ for all $R_i$ through a single scan of $D_c$. Because $c$ was infrequent before the update, the size of $D_c$ is at most $n^{1/3} + 1$. Computing these frequencies therefore takes $O(n^{1/3})$ time. Inserting $c$ and its frequencies into the trees $T_i$ then takes $O(n^{1/3} \log n)$ time. If $c$ was frequent before the update and stayed frequent, we can update the trees $T_i$ for which the updated index is in $A[0 : R_i]$ in $O(n^{1/3} \log n)$ time in total as well. Adding everything up, we can insert and delete a color at a given index in $A$ in $O(n^{2/3})$ time.

### 3.4 Handling insertions and deletions

We now show how to handle insertions and deletions while letting the size of $A$ be variable rather than fixed. To be able to handle insertions, we again work with $n^{1/3}$ blocks, each of size $n^{2/3}$, but this time, we maintain each block as its own array. This is done because we want to make the arrays for each block have a size of $2n^{2/3}$, to have room for inserting colors into a block. We will refer to the elements of each array that have no color in them as *empty* elements.

We will write $A$ to mean the concatenation of these blocks (in order), including the empty elements. Note that while the data structure in Section 3.1 does not take into account these empty elements, we can convert between indices that do take empty elements into account to indices that do not take them into account in $O(n^{1/3})$ time, by keeping track of how many empty elements each block contains and scanning over the blocks.

Assume that before an insertion, each block has at least one empty element. When we insert a color $c$ at a given index $i$ which does not take into account empty elements, we can determine what block $c$ is inserted into in $O(n^{1/3})$ time by scanning over all blocks. We can then determine the position in the block at which $c$ needs to be inserted. Every element in the block after this position is pushed back one position. This requires updating the arrays $D_{c'}$, which takes $O(1)$ time per update [30], totalling $O(n^{2/3})$ time. The index at which we insert $c$ is now open, so we can use the update algorithm in Section 3.3 to insert $c$ into this empty spot. This takes $O(n^{2/3})$ time. Deleting a color works similarly, and takes $O(n^{2/3})$ time as well.

To make sure that every block has at least one empty element before each insertion, we can simply rebuild the data structure once a block is full. Similarly, to make sure that the the number of empty elements in $A$ is not too big, to keep the space complexity $O(n)$, we also rebuild the data structure once a block has less than a quarter of its elements non-empty. These rebuilds happen after at least $n^{2/3}/2$ updates. As the construction time of the data structure was $O(n^{4/3})$, the amortized update time now becomes $O(n^{2/3})$. El-Zein et al. [30] show how to make this amortized time bound worst-case through a more involved insertion and deletion algorithm.

Combined with the result at the beginning of Section 3, we obtain the following result for the chromatic $k$-nearest neighbors problem.

**Theorem 3.3.** *We can build a dynamic data structure of $O(n)$ size, that answers chromatic $k$-nearest neighbors queries on $P$ under the $L_1$, $L_2$ and $L_\infty$ metrics in $O(n^{2/3})$ time, and which supports insertions and deletions in $O(n^{2/3})$ worst-case time.*

## 4 The one-dimensional approximate problem

In this section, we discuss a data structure for solving the approximate one-dimensional problem, and show that by turning to approximations, we can get much lower query time complexities than those stated in the

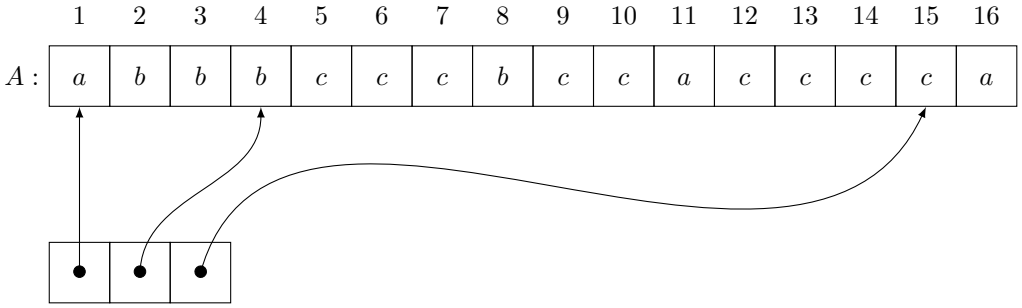



Figure 6: A lookup table for index $i = 1$ for array $A$. The approximation factor is $\varepsilon = 1/2$. Element $a$ is an approximate mode for $A[1:3]$, but in $A[1:4]$, element $b$ has a frequency of 3, more than double that of $a$. Element $b$ is therefore stored next in the lookup table. This element is an approximate mode until range $A[1:15]$, where the frequency of $c$ will be more than double that of $b$.

(conditional) lower bound from Section 2.3. Recall the following definition of the approximate one-dimensional problem.

| ONE-DIMENSIONAL APPROXIAMTE CHROMATIC $k$-NEAREST NEIGHBORS | |
|---|---|
| **Given:** | A set of $n$ colored points $P \subset \mathbb{R}$ in general position, and an approximation factor $\varepsilon \in (0,1)$. |
| **Problem:** | Preprocess $P$ into a data structure, such that given a query point $q \in \mathbb{R}$ and integer $k \in [1,n]$, a color occurring at least $(1-\varepsilon)$ times as often as the most frequent color among the $k$-nearest neighbors of $q$ can be found efficiently. |

In Section 2.1 we gave a data structure that can reduce the problem of (exact) chromatic $k$-nearest neighbors to the range mode problem in an array in $O(\log n)$ time. This reduction is already "fast enough," and does not need to be approximated. We therefore focus on the approximate version of the range mode problem in arrays. We are given an array $A[1:n]$ and an error parameter $\varepsilon \in (0,1)$, and wish to build a data structure on $A$ such that given two indices $1 \leq \ell \leq r \leq n$, we can quickly report an element in $A[\ell:r]$ that occurs at least $(1-\varepsilon)$ times as often as the mode element in $A[\ell:r]$.

## 4.1 An initial data structure

We present the solution of Bose et al. [16]. Their data structure is based on the observation that if we look at all ranges $A[i:j]$ for some fixed $i$, there are only $O(\log_{\frac{1}{1-\varepsilon}}(n-i))$ different query solutions that we need to have ready. This is because if we have an (exact) mode $m$ for some range starting at index $i$, then it will be an approximate mode for all ranges $A[i:j]$ where no element has frequency higher than $1/(1-\varepsilon)$ times the frequency of $m$. In particular, an exact mode $m$ with frequency $f_m$ will be an approximate mode for all ranges $A[i:j]$ where $i \leq j \leq i + f/(1-\varepsilon)$. As $A[i]$ is trivially a mode for range $A[i:i]$ with frequency 1, this gives that in the worst case, we need an approximate mode for the ranges $A[i:j]$ where $j = \lceil i + 1/(1-\varepsilon)^k \rceil$ for some $k$. As there are $O(\log_{\frac{1}{1-\varepsilon}}(n-i))$ such ranges, we need only $O(\log_{\frac{1}{1-\varepsilon}}(n-i))$ approximate modes.

The data structure is a set of $n$ lookup tables, one for each index of $A$. The lookup table for index $i$ stores the $O(\log_{\frac{1}{1-\varepsilon}} n)$ different approximate range modes for the ranges $A[i:j]$, in the form of pointers (indices) to the respective element in $A$. A single lookup table can be constructed in $O(n)$ time with a linear scan of $A$. To build a lookup table for index $i$, keep track of the frequency of every element in $A[i:j]$ while increasing $j$ from $i$ to $n$. Also keep track of the frequency $f_m$ of the exact mode, as well as the frequency $f_a$ of the current approximate mode. Starting at $A[i]$, increment the counter of every element encountered, updating $f_m$ if necessary. Once $f_m > f_a/(1-\varepsilon)$, element $A[j]$ will be the new approximate mode. Therefore add the current index $j$ to the lookup table and set $f_a \leftarrow f_m$. See Figure 6 for an illustration of a lookup table.

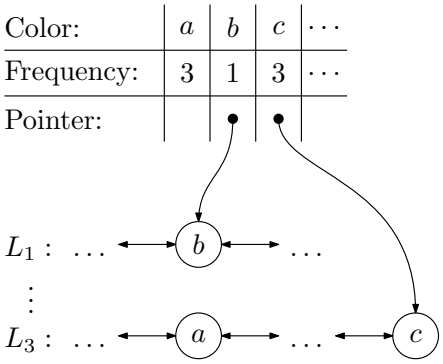



Figure 7: The data structure used during the construction of the tables.

As there are $O(n)$ lookup tables to build, this construction takes $O(n^2)$ time. The total space is $O(n \log_{\frac{1}{1-\varepsilon}} n)$, and the data structure can answer queries in $O(\log \log_{\frac{1}{1-\varepsilon}} n)$ time through binary search in the appropriate lookup table. This results in the following theorem.

**Theorem 4.1.** *Let $A$ be an array of size $n$, and let $\varepsilon \in (0, 1)$ be an approximation factor. In $O(n^2)$ time, we can build a data structure of $O(n \log_{\frac{1}{1-\varepsilon}} n)$ size, that answers approximate range mode queries on $A$ in $O(\log \log_{\frac{1}{1-\varepsilon}} n)$ time.*

## 4.2 Improving the data structure using persistence

Bose et al. [16] managed to get both the preprocessing time and space usage of their data structure down using persistent data structures. We discuss their solution here.

The key idea behind the improvement is to be more precise in what colors we choose as approximate modes. In the initial structure, we stored color $A[i]$ in $T_i[1]$, and afterwards set $T_i[k]$ to be the first color in some range $A[i : j]$ to have a frequency higher than $1/(1 - \varepsilon)$ times the frequency of $T_i[k - 1]$ in the range. For the improvement, we set the following thresholds:

$$f_{\text{high}_1} = 1, \quad f_{\text{high}_k} = f_{\text{low}_{k-1}}/(1 - \varepsilon) + 1,$$
$$f_{\text{low}_1} = 1, \quad f_{\text{low}_k} = f_{\text{high}_{k-1}} + 1.$$

We then again store the index $i$ in $T_i[1]$, but set $T_i[k]$ to the index $j$ where $A[j]$ is the first color in the range $A[i : j]$ to hit a frequency of $f_{\text{high}_k}$. Then, we fix $j$ as the left end of the range, and set $T_{i'}[k] \leftarrow j$ for all $i' > i$ for which the frequency of $A[j]$ in $A[i' : j]$ is at least $f_{\text{low}_k}$. To see that this is correct, consider a query range $A[\ell : r]$. To answer a query, we binary search through $T_\ell$ to find the highest index $j$ that is at most $r$. Color $A[j]$ has a frequency of at least $f_{\text{low}_k}$, for some $k$, while any other color has a frequency of at most $f_{\text{high}_k} - 1 = f_{\text{low}_{k-1}}/(1 - \varepsilon) \leq f_{\text{low}_k}/(1 - \varepsilon)$. The color $A[j]$ is therefore an approximate mode for the range $A[\ell : r]$.

To store the tables, we use persistent search trees [27], storing each table in a different tree. These trees allow for queries and updates in $O(\log m)$ time, where $m$ is the number of entries in a tree, and the storage space used is $O(1)$ per update. Because $f_{\text{high}_k} > f_{\text{high}_{k-1}}/(1 - \varepsilon)$, it follows that there are no more than $\log_{\frac{1}{1-\varepsilon}} n$ rows. The query and update times of the trees are therefore $O(\log \log_{\frac{1}{1-\varepsilon}} n)$.

Constructing the tables works as follows. We construct each row $k$ for all tables before moving on to row $k + 1$. During the construction, we keep track of the frequencies of all colors in some varying range. To do this, we keep a set of $n+1$ doubly-linked lists $L_i$, such that $L_f$ stores the colors with frequency $f$. By keeping pointers from a color to the node in the linked list that stores the color, we can add and remove colors from the lists in $O(1)$ time. In particular, we can increment or decrement the frequency of a color using this data structure in $O(1)$ time. See Figure 7 for an illustration of this data structure.

Let $k$ be the row that is currently being built. Initially, we set $i \leftarrow 1$. We let $j$ go from 1 to $n$, incrementing the frequency of $A[j]$ at each step. Once color $A[j]$ has a frequency of $f_{\text{high}_k}$ in the range $A[i : j]$, we store index $j$ at $T_i[k]$. Then we increment $i$, as row $k$ of table $T_i$ is done, and decrement the frequency of $A[i]$

to reflect this change. While the frequency of $A[j]$ is at least $f_{\text{low}_k}$ in $A[i:j]$, we keep incrementing $i$. Afterwards, we start incrementing $j$ again, continuing the procedure until row $k$ is constructed for all tables.

For each range $A[i:j]$, we perform a constant number of operations when building row $k$ for table $T_i$, aside from potentially adding an entry to the correct persistent tree. Therefore, we spend $O(\log\log_{\frac{1}{1-\varepsilon}} n)$ time per range. Because we increment either $i$ or $j$ after each range that we consider, there are only $O(n)$ ranges considered. The total time taken to build row $k$ for all tables is therefore $O(n\log\log_{\frac{1}{1-\varepsilon}} n)$. Because the number of rows is at most $\log_{\frac{1}{1-\varepsilon}} n$, it follows that the total preprocessing time is $O(n\log_{\frac{1}{1-\varepsilon}} n)$.

To bound the space used by the data structure, we need to bound the number of updates that happen to the tables during construction. The first row of table $T_i$ is always equal to $i$, so this row changes for each table. Now, say we have just set $T_i[k]$ to index $j$, for some $i, j$ and $k \geq 2$, and assume that the frequency of $A[j]$ in $A[i:j]$ is equal to $f_{\text{high}_k}$. Then for all $i' > i$ where the frequency of $A[j]$ is at least $f_{\text{low}_k}$ in $A[i':j]$, we do not have to set entry $T_{i'}[k]$. The number of tables that have row $k$ equal to $j$ is then at least $f_{\text{high}_k} - (f_{\text{low}_k} - 1) = f_{\text{high}_k} - f_{\text{high}_{k-1}} = 1/(1-\varepsilon)^{\lfloor k/2 \rfloor}$. The number of updates that have to be performed for a single row $k$ is then at most $(1-\varepsilon)^{\lfloor k/2 \rfloor} n$, bounding the total number of updates by

$$O\left(n + \sum_{k=2}^{\log_{\frac{1}{1-\varepsilon}} n} (1-\varepsilon)^{\lfloor k/2 \rfloor} n\right) = O\left(n/\varepsilon\right).$$

The space used by the tables will therefore be $O(n/\varepsilon)$.

We summarize the result in the following theorem.

**Theorem 4.2.** *Let $A$ be an array of size $n$, and let $\varepsilon \in (0,1)$ be an approximation factor. In $O(n\log_{\frac{1}{1-\varepsilon}} n)$ time, we can build a data structure of $O(n/\varepsilon)$ size, that answers approximate range mode queries on $A$ in $O(\log\log_{\frac{1}{1-\varepsilon}} n)$ time.*

Combined with the result of Section 2.1, we obtain the following result for the chromatic $k$-nearest neighbors problem.

**Theorem 4.3.** *Let $P$ be a set of $n$ points in $\mathbb{R}$ and let $\varepsilon \in (0,1)$. In $O(n\log_{\frac{1}{1-\varepsilon}} n)$ time, we can build a data structure of $O(n/\varepsilon)$ size, that answers approximate chromatic $k$-nearest neighbors queries on $P$ under the $L_1$, $L_2$ and $L_\infty$ metrics in $O(\log n + \log\log_{\frac{1}{1-\varepsilon}} n)$ time.*

# 5 The two-dimensional problem

In this section we give data structures for the static, exact problems, which were defined as follows.

| TWO-DIMENSIONAL CHROMATIC $k$-NEAREST NEIGHBORS | |
|---|---|
| **Given:** | A set of $n$ colored points $P \subset \mathbb{R}^2$ in general position. |
| **Problem:** | Preprocess $P$ into a data structure, such that given a query point $q \in \mathbb{R}$ and integer $k \in [1,n]$, the most frequent color among the $k$-nearest neighbors of $q$ can be found efficiently. |

We first give data structures that can find a range containing the $k$-nearest neighbors of a query point. Section 5.1 presents a data structure for when the $L_2$ metric is used, and Section 5.2 handles the problem under the $L_1$ and $L_\infty$ metrics. Then in Section 5.3, we give data structures that can find the mode color inside the reported ranges, which leads to our results for the chromatic $k$-nearest neighbors problem.

## 5.1 Finding the $k$-nearest neighbors under the $L_2$ metric

### 5.1.1 Transforming the problem

In this section, we show how to transform the set of two-dimensional points $P$ to a set of planes in $\mathbb{R}^3$, such that the $k$-nearest neighbors of $q$ correspond to the planes lying below a certain point.

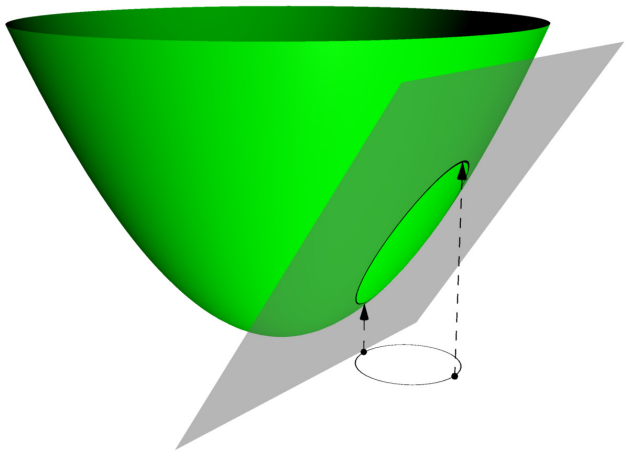


Figure 8: An illustration of the lifting map.

Because the unit circle under the $L_2$ metric is the shape of a circle, there are disks, centered at $q$, that contain precisely the $k$-nearest neighbors of $q$. Let $C_q$ be the circle bounding the smallest of these disks, with $r_q$ its radius.

The first transformation we will apply is a lifting map. Such a map is commonly used for other range searching problems involving circular ranges, like the regular $k$-nearest neighbors problem under the $L_2$ metric. The particular lifting map used takes the two-dimensional points to the 3-dimensional unit paraboloid. This lifting map $\phi : \mathbb{R}^2 \to \mathbb{R}^3$ is defined as

$$\phi : (x, y) \mapsto (x, y, x^2 + y^2).$$

Under this transformation, a circle with center $m$ and radius $r$ is taken to the plane

$$h(m, r) : z = 2m_x x + 2m_y y - m_x^2 - m_y^2 + r^2,$$

and a point in $\mathbb{R}^2$ is mapped below this plane if and only if it lies in the circle. See Figure 8 for an illustration. This property is well known, but for convenience, we prove it in the following lemma.

**Lemma 5.1.** *Let $C$ be a circle with center $m$ and radius $r$. For all points $p \in \mathbb{R}^2$ we have that $\phi(p)$ lies below $h(m, r)$ if and only if $p$ lies in $C$.*

**Proof.** Let $p$ be a point inside $C$ and let $p' = \phi(p)$. Then we have that

$$(p_x - m_x)^2 + (p_y - m_y)^2 \leq r^2,$$

which implies that

$$p_x^2 + p_y^2 \leq 2m_x p_x + 2m_y p_y - m_x^2 - m_y^2 + r^2.$$

By using that $p' = (p_x, p_y, p_x^2 + p_y^2)$, we get that

$$p'_z \leq 2m_x p'_x + 2m_y p'_y - m_x^2 - m_y^2 + r^2,$$

which shows that $p' = \phi(p)$ lies below $h(m, r)$.

By following the above reasoning backwards, it can be seen that if $\phi(p)$ lies below $h(m, r)$ for some point $p \in \mathbb{R}^2$, then $p$ lies in $C$. This proves the theorem. □

In particular, we now have that the points in $C_q$ are taken below the plane $h_q = h(q, r_q)$, and points strictly outside $C_q$ are mapped strictly above $h_q$. Using the fact that $C_q$ bounds the smallest disk with center $q$ that contains the $k$-nearest neighbors of $q$, we get the following corollary.

**Corollary 5.2.** *The plane $h_q = h(q, r_q)$ is the lowest plane of the form $h(q, r)$ such that at least $k$ points from $\phi(P)$ lie below it.*

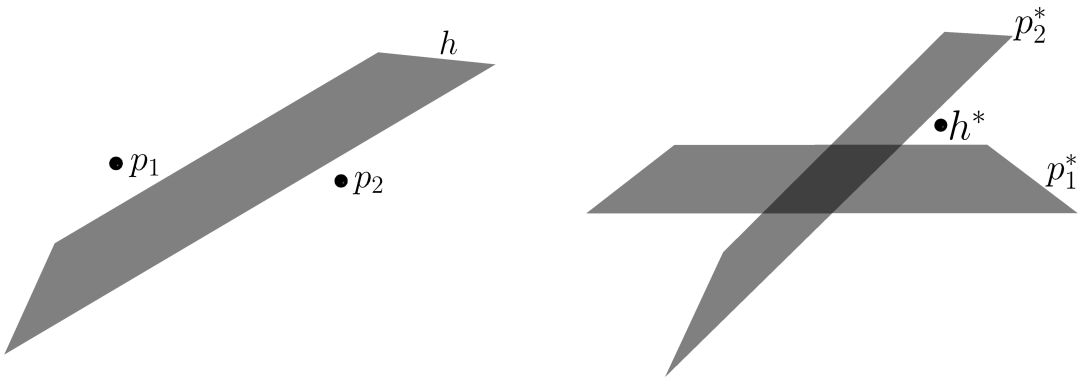



Figure 9: An illustration of duality. The points and plane on the left are dualized to the planes and point on the right.

The problem of finding the mode color among the $k$-nearest neighbors of $q$ has now transformed to that of finding the mode color among the lifted points below the (unknown) plane $h_q$. For the rest of Section 5.1, and later in Section 5.3, we will be working on this problem. To keep notation simple, we will use $P$ to denote the lifted set of points.

The next (and final) transformation that we will apply is a dual transformation. This transformation dualizes a point $p \in \mathbb{R}^3$ to the plane

$$p^* : z = -p_x x - p_y y + p_z,$$

and dualizes a plane

$$h : z = a_x x + a_y y + a_z$$

to the point $h^* = (a_x, a_y, a_z)$. See Figure 9 for an illustration of duality. Note that this duality transformation differs from the usual duality transformation, in which the right side of the equation for $p^*$, and the $z$-coordinate of $h^*$, are multiplied by $-1$. We use this alternative definition of the duality transformation to have the resulting problem be more in line with the concept of levels, which will be used later.

The set $P$ is now dualized to a set of planes $P^*$, and the plane

$$h_q : z = 2q_x x + 2q_y y - q_x^2 - q_y^2 + r_q^2,$$

is dualized to the point $h_q^* = (2q_x, 2q_y, -q_x^2 - q_y^2 + r_q^2)$, where $r_q^2$ is the only unknown term. This dual transformation changes the problem of finding the mode color of the points below $h_q$ to that of finding the mode color among the planes that lie below $h_q^*$. This follows from the following lemma.

**Lemma 5.3.** *Let $h \subset \mathbb{R}^3$ be a non-vertical plane and let $h^*$ be its dual point. Let $p \in \mathbb{R}^3$ be a point, with $p^*$ its dual plane. Then $p$ lies below $h$ if and only if $p^*$ lies below $h^*$. Furthermore, point $p$ lies strictly below $h$ if and only if $p^*$ lies strictly below $h^*$.*

**Proof.** Let $h : z = a_x x + a_y y + a_z$ be a plane, with $h^*$ its dual point. Let $p \in \mathbb{R}^3$ be a point, with $p^*$ its dual plane. We have that $p$ lies below $h$ if and only if

$$p_z \leq a_x p_x + a_y p_y + a_z,$$

which rewrites to

$$a_z \geq -p_x a_x - p_y a_y + p_z.$$

This shows that $p$ lies below $h$ if and only if $p^*$ lies below $h^*$. By changing the inequalities to their strict variants, it can also be shown that $p$ lies strictly below $h$ if and only if $p^*$ lies strictly below $h^*$. □

### 5.1.2 The data structure

In this section, we describe a data structure that can find the point $h_q^*$ in dual space, and therefore the plane $h_q$ in primal space. We can then use the data structure to reduce the chromatic $k$-nearest neighbors problem to the halfspace range mode problem, which is discussed in Section 5.3.

To describe our data structure, we use the concept of levels. The *level* of a point $p$ in an arrangement $\mathcal{A}(H)$ formed by a set of planes $H$, is the number of planes in $H$ that lie strictly below $p$. An important theorem that follows from Lemma 5.3 has to do with levels. It is stated as follows.

**Theorem 5.4.** *Let $h \subset \mathbb{R}^3$ be a non-vertical plane and let $h^*$ be its dual point. Let $h^-$ be the open halfspace bounded above by $h$. The level of $h^*$ in $\mathcal{A}(P^*)$ is equal to $|h^- \cap P|$.*

The relation between the levels of points and our search for the point $h_q^*$ is given in the following theorem. Corollary 5.6 follows from this theorem.

**Theorem 5.5.** *Let $\ell = \{(2q_x, 2q_y, z) \in \mathbb{R}^3 : z \in \mathbb{R}\}$ be a vertical line in dual space. The point $h_q^*$ is the highest point on $\ell$ for which the level in $\mathcal{A}(P^*)$ is at most $k-1$.*

**Proof.** By Corollary 5.2, we have that $h_q$ is the lowest plane of the form $h(q, r)$ that has at least $k$ points from $P$ below it. This implies that the open halfspace bounded above by $h_q$ contains less than $k$ points from $P$. Theorem 5.4 now implies that the level of $h_q^*$ in $\mathcal{A}(P^*)$ is at most $k-1$.

Now let $h^* = (2q_x, 2q_y, -q_x^2 - q_y^2 + r_q^2 + \alpha)$ for some $\alpha > 0$. Then $h^*$ lies on $\ell$ and strictly above $h_q^*$. The dual plane of $h^*$ is

$$h : z = 2q_x x + 2q_y y - q_x^2 - q_y^2 + r^2 + \alpha,$$

which lies strictly above $h_q$. This implies that $h^-$, the open halfspace bounded by $h$, contains at least $k$ points of $P$. Therefore, by Theorem 5.4, the level of $h^*$ will be at least $k$. This proves the theorem. □

**Corollary 5.6.** *The point $h_q^*$ lies on a plane in $P^*$.*

We now describe our data structure. The general idea behind the data structure is to find two points $p^+$ and $p^-$ on the line $\ell = \{(2q_x, 2q_y, z) \in \mathbb{R}^3 \mid z \in \mathbb{R}\}$, with $p^+$ above $h_q^*$ and $p^-$ below $h_q^*$. By making sure that the number of planes between the two points is small, we can go over all these planes naively to find $h_q^*$, while keeping the query-time complexity low.

The main component of our data structure is a $(1/r)$-cutting of $\mathcal{A}(P^*)$. A *cutting* of $\mathbb{R}^3$ is a partitioning of $\mathbb{R}^3$ into (possibly unbounded) full-dimensional simplices with disjoint interiors. A cutting is a *$(1/r)$-cutting* of an arrangement $\mathcal{A}(H)$, for $1 \leq r \leq n$, if the interior of every simplex intersects at most $n/r$ planes of $H$. The current results on these cuttings are summarized in the following theorem.

**Theorem 5.7.** *Given a set of $n$ planes $H$ in $\mathbb{R}^3$, a $(1/r)$-cutting of $\mathcal{A}(H)$ exists that consists of $O(r^3)$ simplices, which is optimal. Such a cutting can be constructed in $O(nr^2)$ time. If $r \leq n^\alpha$ for a constant $\alpha < 1/3$, the construction time can be decreased to $O(n \log r + r^5)$.*

**Proof.** The existence of a $(1/r)$-cutting consisting of $O(r^3)$ simplices, which is asymptotically optimal, was proven by Chazelle and Friedman [23]. Chazelle [21] gave an $O(nr^2)$ time algorithm for constructing such a cutting. The faster construction algorithm for small values of $r$ is due to Matoušek [43]. □

Apart from the cutting, we also need a data structure for halfspace range counting, as well as one that can report all points between two parallel planes. We give a more general data structure for range searching in polyhedra of $O(1)$ size, as we will need it later. Because the intersection of $O(1)$ halfspaces forms such a polyhedron, we can use the following result to obtain a data structure that performs the above two queries.

**Theorem 5.8.** *Let $P$ be a set of $n$ points in $\mathbb{R}^3$. In $O(n^{1+\delta})$ time, we can build a data structure of $O(n)$ size, that can report or count the (number of) points of $P$ in a given convex (possibly unbounded) query polyhedron of $O(1)$ size in $O(n^{2/3} + k)$ time, where $k$ is the output size.*

**Proof.** Let $Q$ be a convex (possibly unbounded) polyhedron of $O(1)$ size. We first show how to partition $Q$ into $O(1)$ simplices, using the cone triangulation of Sleator et al. [49]. For this triangulation, first triangulate all faces of $Q$. This can be done in $O(1)$ time, and will result in $O(1)$ triangles. Now pick a point $q \in Q$.

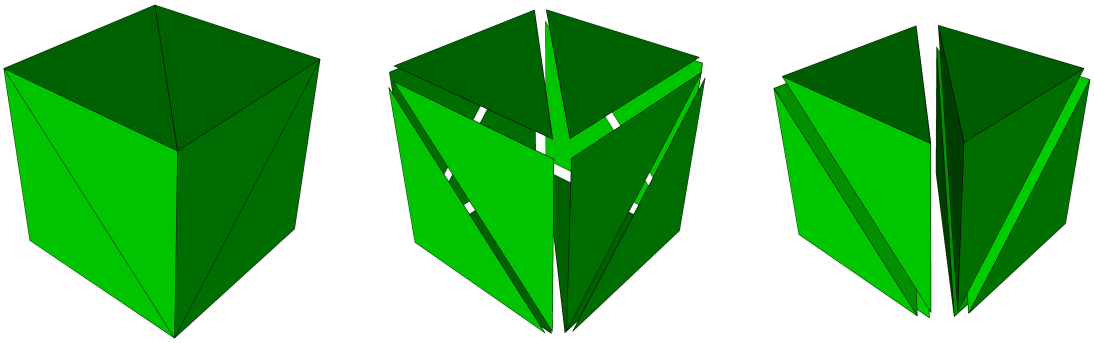


Figure 10: An illustration of a cube partitioned into simplices. The first two images on the left show a possible triangulation of the faces of the cube. The image on the right shows how the triangles are extended to create a partitioning of the cube into simplices, by connecting each triangle to the bottommost vertex in the image.

Note that we do not pick a vertex here, as there might not be any. For each triangle that is not incident to $q$, build a simplex with the triangle as a side, and which contains $q$ as its vertex. See Figure 10 for an illustration. The resulting set of simplices will partition $Q$. These simplices can be constructed in $O(1)$ time, and by Sleator et al. [49], there will be $O(1)$ created simplices.

The theorem now follows from a result of Matoušek [44] on simplex range searching, who showed how to build a data structure of $O(n)$ size in $O(n^{1+\delta})$ time, that can report or count the (number of) points in a query simplex in $O(n^{2/3} + k)$ time, where $k$ is the output size. Given a convex query polyhedron $Q$ of $O(1)$ size, we first partition $Q$ into $O(1)$ simplices, and query the data structure of Matoušek on all simplices, combining the results. Note that points can lie in multiple simplices. To make sure that no point is reported twice, simply mark a reported point as reported, and only report a point if it has not been reported yet. As there are only $O(1)$ simplices that we query with, a point will not be checked more than $O(1)$ times, so this check will not affect the query time complexity. □

**Remark.** The preprocessing time of the range-searching data structure can be lowered to $O(n \log n)$ with the result of Chan [18], though the query times will not be deterministic (they hold "with high probability"). As having a preprocessing time of $O(n^{1+\delta})$ will not change the preprocessing time of our final data structure, we use the result of Matoušek to keep the query time deterministic.

The total preprocessing time of the cutting and range-searching data structure is $O(n^{1+\delta} + nr^2)$, or $O(n^{1+\delta} + r^5)$ if $r \leq n^\alpha$ for some constant $\alpha < 1/3$. The space used is $O(r^3 + n)$.

**The query algorithm.** We now show how to query the data structure, with a query line $\ell$. Let $\Xi$ be a $(1/r)$-cutting of $\mathcal{A}(P^*)$, consisting of $O(r^3)$ simplices. For a simplex $\Delta \in \Xi$, we write $\Delta_\ell$ to mean the set of intersection points between the boundary of $\Delta$ and line $\ell$. We say that the points in the set

$$\Xi_\ell = \bigcup_{\Delta \in \Xi} \Delta_\ell \cup \{(2q_x, 2q_y, -\infty), (2q_x, 2q_y, \infty)\},$$

are *candidate points*. There are $O(r^3)$ candidate points, which can be found in $O(r^3)$ time. Next, we sort these points from lowest to highest, taking $O(r^3 \log r)$ time. We then perform binary search on the sorted points, looking for the highest point that lies under $h_q^*$.

Let $h^* \in \Xi_\ell$ be one of the candidate points. Assume for a moment that the $z$-coordinate of $h^*$ is finite. Then $h^*$ is dual to a non-vertical plane $h$. Therefore, we can use Theorem 5.4 to find the level of $h^*$ using halfspace range counting in primal space, which can be done in $O(n^{2/3})$ time. If the $z$-coordinate of $h^*$ is infinite, the level of $h^*$ is either 0 (when the coordinate is $-\infty$) or $n$ (when the coordinate is $\infty$). If the level is at most $k-1$, we know that $h_q^*$ lies above $h^*$. If the level is higher than $k-1$, we know that $h_q^*$ lies strictly below $h^*$. Using this procedure in a binary-search algorithm, we obtain an algorithm consisting of $O(\log r)$ steps, each taking $O(n^{2/3})$ time, that returns the highest point in $\Xi_\ell$ that lies under $h_q^*$.

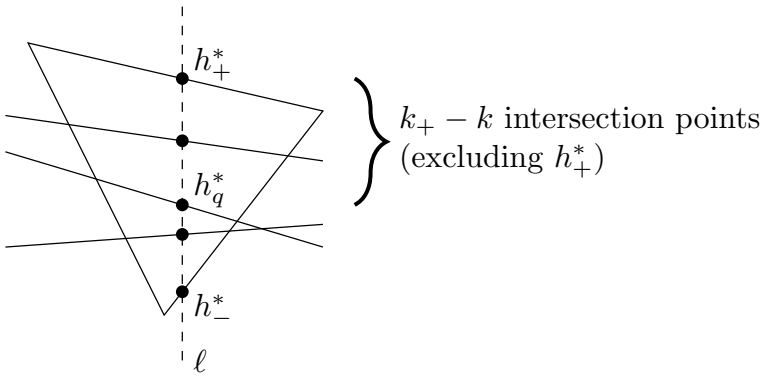



Figure 11: The positions of $h_-^*$, $h_+^*$ and $h_q^*$ within a simplex.

Let $h_-^*$ be the found candidate point, and let $h_+^* \in \Xi_\ell$ be the next candidate point in the sorted order. Assume without loss of generality that $h_-^*$ and $h_+^*$ have finite $z$-coordinates. Let $h_-$ and $h_+$ be their corresponding planes in primal space. Observe that there is a simplex $\Delta \in \Xi_\ell$ with $h_-^*, h_+^* \in \Delta$. Therefore, because $\Delta$ is part of a $(1/r)$-cutting of $\mathcal{A}(P^*)$, the number of planes passing strictly between $h_-^*$ and $h_+^*$ is at most $n/r$. Because the intersection of two halfspaces forms a convex (unbounded) polyhedron of $O(1)$ size, the points between $h_-$ and $h_+$ can be reported using Theorem 5.8. As the planes between $h_-^*$ and $h_+^*$ are dual to these points, it follows that we can report the planes between $h_-^*$ and $h_+^*$ in $O(n^{2/3} + n/r)$ time.

Now, because of how we chose $h_-^*$, we have that $h_q^*$ lies above $h_-^*$ and strictly below $h_+^*$. By Corollary 5.6, we now get that $h_q^*$ lies on one of the reported $O(n/r)$ planes, or it is equal to $h_-^*$. Let $k_+$ be the level of $h_+^*$, which we can find by halfspace range counting in primal space. We can now find $h_q^*$ by sorting the intersection points between $\ell$ and the reported planes from highest to lowest, including duplicate points, and adding $h_-^*$ as well. The $(k_+ - k)$-th intersection point will be equal to $h_q^*$ (see Figure 11). This final step takes $O((n/r) \log n)$ time.

We summarize the result in the following theorem.

**Theorem 5.9.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$ and let $1 \leq r \leq n$. In $O(n^{1+\delta} + nr^2)$ time, we can build a data structure of $O(n + r^3)$ size that, given a query point $q \in \mathbb{R}^2$, finds the smallest disk $D_q$ centered at $q$ which contains precisely the $k$-nearest neighbors of $q$, in*

$$O(r^3 \log r + n^{2/3} \log r + (n/r) \log n)$$

*time. If $r \leq n^\alpha$ for a constant $\alpha < 1/3$, the preprocessing time lowers to $O(n^{1+\delta} + r^5)$.*

## 5.2 Finding the $k$-nearest neighbors under the $L_1$ and $L_\infty$ metrics

We first show how the problem under the $L_1$ metric can be transformed to use the $L_\infty$ metric. Afterwards, we focus solely on the $L_\infty$ metric. The transformation is shown in the following lemma. See also Figure 12 for the intuition behind the transformation.

**Lemma 5.10.** *If we can build a data structure on $P$ of $S(n)$ size in $P(n)$ time, which answers chromatic $k$-nearest neighbors queries under the $L_\infty$ metric in $Q(n)$ time, then we can build a data structure on $P$ of $O(S(n))$ size in $O(P(n)+n)$ time, which answers chromatic $k$-nearest neighbors queries under the $L_1$ metric in $O(Q(n) + 1)$ time.*

**Proof.** Define the map $\phi : \mathbb{R}^2 \to \mathbb{R}^2$ as

$$\phi : (x, y) \mapsto (x - y, x + y).$$

Let $p, q \in \mathbb{R}^2$. We will first show that the distance under the $L_1$ metric between $p$ and $q$ is the same as the distance between $\phi(p)$ and $\phi(q)$ under the $L_\infty$ metric. By the definition of the $L_\infty$ metric, we have that

$$\begin{aligned} d_\infty(\phi(p), \phi(q)) &= \max\{|(p_x - p_y) - (q_x - q_y)|,\ |(p_x + p_y) - (q_x + q_y)|\} \\ &= \max\{|(p_x - q_x) - (p_y - q_y)|,\ |(p_x - q_x) + (p_y - q_y)|\}. \end{aligned}$$

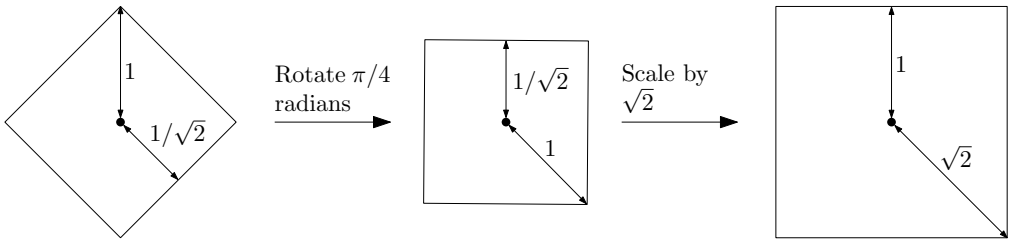



Figure 12: The unit circle under the $L_1$ (left) and $L_\infty$ (right) metrics. The unit circle under the $L_1$ metric is transformed to that under the $L_\infty$ metric by a clockwise rotation of $\pi/4$ radians and scaling the result by $\sqrt{2}$. This transformation is the mapping $\phi : (x, y) \mapsto (x - y, x + y)$.

We distinguish between the following four cases:

(i) If $p_x - q_x \leq 0$ and $p_y - q_y \leq 0$, then

$$\begin{aligned} d_\infty(\phi(p), \phi(q)) &= |(p_x - q_x) + (p_y - q_y)| \\ &= |p_x - q_x| + |p_y - q_y| \\ &= d_1(p, q). \end{aligned}$$

(ii) If $p_x - q_x > 0$ and $p_y - q_y \leq 0$, then

$$\begin{aligned} d_\infty(\phi(p), \phi(q)) &= |(p_x - q_x) - (p_y - q_y)| \\ &= |p_x - q_x| + |p_y - q_y| \\ &= d_1(p, q). \end{aligned}$$

(iii) If $p_x - q_x \leq 0$ and $p_y - q_y > 0$, the proof is analogous to that of (ii).

(iv) If $p_x - q_x > 0$ and $p_y - q_y > 0$, the proof is analogous to that of (i).

It follows that we can use $\phi$ to transform the problem under the $L_1$ metric to that under the $L_\infty$ metric. We can apply the map to any point in $O(1)$ time, and therefore to all points in $P$ in $O(n)$ time. This proves the theorem. □

The unit circle under the $L_\infty$ metric forms an axis-aligned square. Therefore, given a point $q$, there are axis-aligned squares, centered at $q$, that contain precisely the $k$-nearest neighbors of $q$. Let $S_q$ be the smallest of these squares.

We say that the radius of a square is half the length of one of its sides. Define $S(p, r) = [p_x - r, p_x + r] \times [p_y - r, p_y + r]$ as the axis-aligned square with center $p \in \mathbb{R}^2$ and radius $r \geq 0$. The square $S_q$ is then given as $S(q, r_q)$, for some radius $r_q \geq 0$. Because of how we defined $S_q$, we can make the following observation.

**Observation 5.11.** There is a point $p \in P$ such that $r_q = d_\infty(q, p)$ is the radius of $S_q$.

By the definition of the $L_\infty$ metric, we have that if $r_q = d_\infty(q, p)$ for some $p \in P$, then $r_q$ is either equal to $|q_x - p_x|$ or equal to $|q_y - p_y|$. It therefore follows that in order to search for $r_q$, we merely have to search through the $x$- and $y$-coordinates of the points in $P$.

Our data structure for finding $S_q$ is simple. We need two sequences of sorted values. Let $x_1 \leq \cdots \leq x_n$ be the sorted $x$-coordinates of the points in $P$, and similarly let $y_1 \leq \cdots \leq y_n$ be the sorted $y$-coordinates. We also set $x_0 = y_0 = -\infty$ and $x_{n+1} = y_{n+1} = \infty$. Aside from these two sequences, we need a data structure that can count the number of points inside an axis-aligned query square. Good candidates for this are the $k$-d tree[2] [12] and the range tree [13], both introduced by Bentley. A $k$-d tree uses $O(n)$ space, can be build in $O(n \log n)$ time, and answers queries in $O(\sqrt{n})$ time. A range tree uses $O(n \log n)$ space, can be build in

[2]The $k$ in $k$-d tree was originally used to denote the dimension of the stored points. It is therefore not related to the parameter $k$ in the chromatic $k$-nearest neighbors problem.

$O(n \log n)$ time, and answers queries in $O(\log^2 n)$ time. By modifying the range tree, its query time can be lowered to $O(\log n)$ [52].

Let $x_0, \ldots, x_\ell$ be the $x$-coordinates that are at most $q_x$. The sequence $r_i = |q_x - x_i|$, for $i = 0, \ldots, \ell$, defines a sequence of decreasing radii. Therefore, $S(q, r_i) \supseteq S(q, r_j)$ for $i \leq j$. By using the fact that $S_q$ is the smallest axis-aligned square, centered at $q$, that contains the $k$-nearest neighbors of $q$, we can now use a combination of binary search and range counting to find the smallest radius $r_i$ for which $S(q, r_i) \supseteq S_q$. This is done by performing range counting on the current square $S$, and checking whether the count is at least $k$ or not. If it is at least $k$, then we have that $S_q \subseteq S$. Otherwise, we have that $S_q \supseteq S$. The result is a procedure that uses $O(\log n)$ square range-counting queries, and finds the smallest radius $r_i$ for which $S(q, r_i) \supseteq S_q$.

The above procedure can be slightly modified such that it can search through the $x$-coordinates that are greater than $q_x$, and similarly for the $y$-coordinates. The result is a set of four radii that are all at least $r_q$. Note that because the four procedures combined consider all possible distances between $q$ and a point in $P$ as a radius, it follows that $r_q$ must be one of the four returned radii. The smallest of the four radii must now be the correct one. Therefore, we can use the four procedures to find the square $S_q$ in $O(\log n)$ orthogonal range counting queries. This result implies the following theorem.

**Theorem 5.12.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$. In $O(n \log n)$ time, we can build a data structure of $O(n)$ size, that can report an axis-aligned square containing precisely the $k$-nearest neighbors of a query point in $O(\sqrt{n} \log n)$ time. Alternatively, with $O(n \log n)$ space, the query time can be lowered to $O(\log^2 n)$.*

## 5.3 Range mode queries

In this section, we give data structures that can compute the mode color inside the ranges reported with the data structures of Sections 5.1 and 5.2. Section 5.3.1 first presents the general data structure that is used for these queries. Then in Sections 5.3.2 and 5.3.3, we analyze the implementation details and the performance of this general data structure, for the problem under the $L_2$ metric and under the $L_1$ and $L_\infty$ metrics, respectively. Finally, in Section 5.3.4, we give a more involved preprocessing algorithm for the data structure, which leads to better complexities for our problems.

### 5.3.1 Range mode queries in arrangements of general surfaces

Chan et al. [20] describe a data structure that can perform halfspace range mode queries, which is equivalent to the dual problem of preprocessing a set of planes, such that the mode color among the planes that lie below a query point can be found efficiently. This data structure can immediately be used in combination with our data structure from Section 5.1 to answer chromatic $k$-nearest neighbors queries under the $L_2$ metric. However, we can generalize their data structure to work on arrangements of general $xy$-monotone surfaces, and then use this generalized data structure for the problem under the $L_1$ and $L_\infty$ metrics as well. Because the analysis of the data structure will differ between metrics, we give the data structure in this section, and analyze it in Sections 5.3.2 and 5.3.3.

**Remark.** Aside from a data structure for halfspace range mode queries, Chan et al. [20] also give a data structure for range mode queries in orthogonal ranges. While this data structure can be used with our data structure from Section 5.2, its complexities are worse than what we will achieve for our problem, where the ranges are axis-aligned squares rather than more general orthogonal ranges.

We first generalize the concept of $(1/r)$-cuttings to work on general surfaces. To this end, we define a $(1/r)$-cutting of an arrangement $\mathcal{A}(S)$ of a set of $n$ surfaces $S$ to be a cutting of $\mathbb{R}^3$ that has the additional property that the interior of each simplex in the cutting is intersected by at most $n/r$ surfaces of $S$. Now let $S$ be a given set of $n$ $xy$-monotone surfaces that we have to preprocess into a data structure. The data structure consists of a $(1/r)$-cutting $\Xi$ of $\mathcal{A}(S)$, paired with a point-location data structure on $\Xi$. For each simplex $\Delta \in \Xi$, we store the mode color among those surfaces of $S$ that lie strictly below $\Delta$.

Alongside the cutting, we need a data structure that can report the surfaces that intersect a given simplex (not necessarily in its interior), and which lie below a query point inside the simplex. Finally, for each color $i$, we use a data structure that can count the number of surfaces with color $i$ that lie below a given query point.

Querying the structure with a query point $q$ works by first locating a simplex $\Delta \in \Xi$ that contains $q$. The surfaces below $q$ all either lie strictly below $\Delta$, or they intersect $\Delta$. Using Observation 2.4 by Krizanc et al. [38], this leads to the following observation.

**Observation 5.13.** The mode color among the surfaces below $q$ is either the color stored in $\Delta$, or it is one of the colors of the surfaces below $q$ that intersect $\Delta$.

The query algorithm now proceeds as follows. First, all at most $n/r$ surfaces below $q$ that intersect $\Delta$ are reported. Let $C_\Delta$ be the different colors of the reported surfaces. Then, for each color $c \in C_\Delta$, the number of surfaces below $q$ that have color $c$ is counted. By also counting the number of surfaces below $q$ that have the color that is stored in $\Delta$, the mode color can be found by picking a color whose count is the highest.

### 5.3.2 Range mode analysis under the $L_2$ metric

We now analyze the complexities of the data structure from Section 5.3.1 for arrangements of planes, using the analysis of Chan et al. [20]. Recall that this data structure can be used in combination with that of Section 5.1 to solve the chromatic $k$-nearest neighbors problem under the $L_2$ metric.

Let $H$ be a given set of $n$ planes in $\mathbb{R}^3$ that lie in general position. The data structure, when built on $H$, consists of a $(1/r)$-cutting of $\mathcal{A}(H)$, paired with a point-location data structure. A $(1/r)$-cutting of optimal $O(r^3)$ size, together with a point-location data structure that answers queries in $O(\log r)$ time, can be constructed in $O(nr^2)$ time using the result of Chazelle [22]. Let $\Xi$ be the constructed cutting. For each simplex $\Delta \in \Xi$, we can compute the mode color among those planes in $H$ that lie strictly below $\Delta$ through a linear scan of $H$, taking $O(n)$ time per simplex. The total construction time for this part of the data structure is therefore $O(nr^3)$, and the total space used is $O(r^3)$.

**Remark.** The algorithm of Chazelle [22] for constructing cuttings uses $O(nr^2)$ space during construction. As an alternative, the cuttings algorithm of Matoušek [43] can be used to create a $(1/r)$-cutting of $O(r^3)$ size using $O(n \log r)$ space during construction. We can then construct a point-location data structure on this cutting as described in [50]. The resulting point-location data structure uses $O(r^3 \log r)$ space. One thing to note here is that the algorithm of Matoušek requires that $r \leq n^{1/3-\delta}$ for a constant $\delta > 0$. This will lead to an extra factor $O(n^\delta)$ in the query time of the data structure for halfspace range mode queries.

The data structure that can report the planes that intersect a given simplex, and which lie below a query point inside the simplex, uses Theorem 5.8 for reporting points in polyhedra, and is given in the following theorem.

**Theorem 5.14.** *In $O(n^{1+\delta})$ time, we can build a data structure of $O(n)$ size, that can report the planes of $H$ that intersect a given simplex, and which lie below a given point inside the simplex, in $O(n^{2/3} + k)$ time, where $k$ is the number of reported planes and $\delta > 0$ is an arbitrarily small constant.*

**Proof.** The data structure that we build is that of Theorem 5.8. This data structure can be built in $O(n^{1+\delta})$ time, uses $O(n)$ space, and can report the $k$ points in a given convex polyhedron of $O(1)$ size in $O(n^{2/3} + k)$ time. We build this data structure on the points $H^*$ that are dual to the planes of $H$.

The planes intersecting $\Delta$ are precisely those that intersect an edge of $\Delta$. Fix an edge $e = (v, w)$ of $\Delta$. The set of planes intersecting $e$ consists of those planes that lie above one vertex of $v$ and $w$, and below the other. Let $H_e$ be the set of planes intersecting $e$, with $H_e^*$ its dual set of points. Let $v^*$, respectively $w^*$, be the dual planes of $v$, respectively $w$. By Lemma 5.3, it must be that all points in $H_e^*$ lie above one of $v^*$ and $w^*$, and below the other. Because the intersection of two halfspaces forms a convex (unbounded) polyhedron of $O(1)$ size, it follows from Theorem 5.8 that the points above $v^*$ but below $w^*$, as well as those below $v^*$ but above $w^*$, can be reported in $O(n^{2/3} + k_e)$ time, where $k_e$ is the number of points that are reported. In the same time, we can report the planes dual to these points, which are the planes intersecting $e$.

To report all planes intersecting an edge of $\Delta$, we perform the above procedure for all edges of $\Delta$. Because a plane can only intersect a constant number of edges, the total number of planes reported is $O(k)$. To prevent a plane from being reported multiple times, simply mark a reported plane as reported, and only report a plane when it has not been marked as reported yet. In total, these checks take $O(k)$ extra time. It follows that we can report all planes intersecting $\Delta$ in $O(n^{2/3}+k)$ time. In $O(k)$ extra time, we can filter the reported planes to only include those below a given query point. □

The number of planes intersecting a simplex in $\Xi$ is $O(n/r)$. This is because we assumed the planes of $H$ to lie in general position, which means that the number of planes that intersect the boundary of $\Delta$, but not the interior of $\Delta$, is $O(1)$. This leads to the following corollary.

**Corollary 5.15.** *In $O(n^{1+\delta})$ time, we can build a data structure of $O(n)$ size, that reports the planes of $H$ that intersect a simplex $\Delta \in \Xi$, and which lie below a given point in $\Delta$, in $O(n^{2/3} + n/r)$ time, where $\delta > 0$ is an arbitrarily small constant.*

The final part of the data structure is the set of range-counting data structures for the different sets of planes with equal colors. For each color $c$, let $H_c$ be the set of planes with color $c$. Counting the number of planes below a query point that have color $c$ is equivalent to counting the number of points dual to $H_c$ that lie in a query halfspace. We therefore build a data structure for halfspace range counting on each set of points $H_c^*$ that is dual to $H_c$. Using the result of Matoušek [44], we can build all these data structures in

$$O\left(\sum_c |P_c|^{1+\delta}\right) = O(n^{1+\delta})$$

time in total. The data structures use a total of

$$O\left(\sum_c |P_c|\right) = O(n)$$

space, and can answer a query on color $c$ in $O(|P_c|^{2/3})$ time. The total preprocessing time of the data structure is therefore $O(n^{1+\delta} + nr^3)$ and the total space used is $O(n + r^3)$.

The time taken to answer a query is $O(\log r)$ to perform a point-location query, resulting in a simplex $\Delta$ being reported that contains the query point. Reporting the planes intersecting $\Delta$ can then be done in $O(n^{2/3} + n/r)$ time, by Corollary 5.15. Let $C_\Delta$ be the set of different colors of the reported planes. The final step was to count the frequencies of these colors among the planes below the query point, as well as counting the frequency of the color stored in $\Delta$. This step takes

$$O\left(\sum_{c \in C_\Delta} |P_c|^{2/3} + n^{2/3}\right)$$

time.

Because $C_\Delta$ will contain at most $O(n/r)$ colors, we can assume without loss of generality that $C_\Delta = \{1, \ldots, O(n/r)\}$. Hölders inequality states that

$$\left(\sum_{i=1}^{m} x_i^a y_i^b\right)^{a+b} \leq \left(\sum_{i=1}^{m} x_i^{a+b}\right)^a \left(\sum_{i=1}^{m} y_i^{a+b}\right)^b,$$

for any two sequences $x_1, \ldots, x_m$ and $y_1, \ldots, y_m$, and any two positive values $a$ and $b$. This inequality now implies that

$$\left(\sum_{c \in C_\Delta} |P_c|^b\right)^{a+b} \leq \left(\sum_{c \in C_\Delta} 1^{a+b}\right)^a \left(\sum_{c \in C_\Delta} |P_c|^{a+b}\right)^b = O\left((n/r)^a \cdot \left(\sum_{c=1}^{n/r} |P_c|^{a+b}\right)^b\right)$$

for all positive values $a$ and $b$. If we now set $a = 1/3$ and $b = 2/3$ , it follows that the range counting queries take

$$\begin{aligned} O\left(\sum_{c \in C_\Delta} |P_c|^{2/3} + n^{2/3}\right) &= O\left((n/r)^{1/3} \cdot n^{2/3} + n^{2/3}\right) \\ &= O(n/r^{1/3}), \end{aligned}$$

time. This also bounds the total query time.

We summarize the results in the following theorem.

**Theorem 5.16.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$ and let $1 \leq r \leq n$. In $O(n^{1+\delta} + nr^3)$ time, we can build a data structure of $O(n + r^3)$ size, that answers halfspace range mode queries in $O(n/r^{1/3})$ time, where $\delta > 0$ is an arbitrarily small constant.*

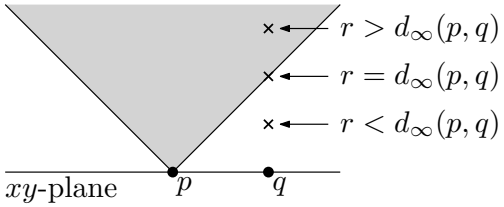



Figure 13: An illustration of the upside-down pyramid $\nabla_p$. The distance between $p$ and $q$ under the $L_\infty$ metric (in $\mathbb{R}^2$) determines whether the point $(q_x, q_y, r)$ lies strictly under, on, or strictly above $\nabla_p$.

### 5.3.3 Range mode analysis under the $L_1$ and $L_\infty$ metrics

We can use the data structure from Section 5.3.1 for the problem under the $L_1$ and $L_\infty$ metrics as well. Recall from Section 5.2 that we can focus solely on the $L_\infty$ metric.

For a two-dimensional point $p$, the graph of its distance function

$$d_\infty(p, q) = \max\{|p_x - q_x|, |p_y - q_y|\}$$

forms an upside-down pyramid $\nabla_p$ in $\mathbb{R}^3$. If we now have a two-dimensional point $q$, then we have that $d_\infty(p, q) \leq r$ if and only if $(q_x, q_y, r)$ lies above $\nabla_p$. See Figure 13 for an illustration. Thus, by looking at the graphs of the distance functions for each point in $P$, the problem is transformed to that of finding the mode color among those graphs in $\mathbb{R}^3$ that lie below a given query point. Because these graphs are $xy$-monotone surfaces, we can use the data structure from Section 5.3.1 for this problem.

We write $\nabla_p$ to denote the graph $d_\infty(p, q)$ for a point $p \in \mathbb{R}^2$, and write $\nabla = \{\nabla_p \mid p \in P\}$ to be the set of graphs for all points in $P$. For the data structure, we need to construct a $(1/r)$-cutting for the arrangement $\mathcal{A}(\nabla)$ formed by $\nabla$. The following theorem shows that we can use cuttings for planes to construct the desired cuttings.

**Theorem 5.17.** *Let $1 \leq r \leq n$. In $O(nr^2)$ time, we can construct $(1/r)$-cutting of $\mathcal{A}(\nabla)$ that consists of $O(r^3)$ simplices. With $O(r^3)$ additional preprocessing and space, a data structure for point location in the cutting can be build which answers queries in $O(\log r)$ time.*

**Proof.** Let $\nabla_p$ be a pyramid in $\nabla$. This pyramid is contained in the union of the four planes

$$p_x = x \pm z,$$
$$p_y = y \pm z.$$

Now let $H$ be the set of all $4n$ (not necessarily distinct) planes whose union contains all of $\nabla$. Let $\Xi$ be a $(1/(4r))$-cutting of $\mathcal{A}(H)$. If the interior of a simplex $\Delta \in \Xi$ intersects a pyramid in $\nabla$, it must also intersect a plane in $H$. Therefore, the number of pyramids in $\nabla$ that intersect the interior of $\Delta$ is bounded by the number of planes in $H$ that intersect the interior of $\Delta$, which is at most $4n/(4r) = n/r$. The cutting $\Xi$ is therefore a $(1/r)$-cutting for $\mathcal{A}(\nabla)$.

If the planes of $H$ are in general position, we can use a result by Chazelle [22] to finish the proof. Unfortunately, the planes are definitely not in general position. For one, the planes constructed for a single pyramid $\nabla_p$ all contain $p$, meaning that four planes intersect in a single point. Moreover, for any two points $p_1$ and $p_2$, we have that the planes constructed for $\nabla_{p_1}$ and $\nabla_{p_2}$ form four pairs of parallel (or even coinciding) planes. Luckily, we can perturb the set of planes $H$ such that they do lie in general position, while keeping the properties of the constructed cutting intact [29, 31]. That is, the $(1/(4r))$-cutting constructed for the perturbed planes will be a $(1/r)$-cutting for $\mathcal{A}(\nabla)$. The theorem now follow from [22]. □

Let $\Xi$ be the constructed $(1/r)$-cutting of $\mathcal{A}(\nabla)$. For each simplex $\Delta \in \Xi$, we can compute the mode color among the pyramids strictly below $\Delta$ in $O(n)$ time through a linear scan over $\nabla$. This results in a total construction time for the cutting of $O(nr^3)$.

To report the pyramids that intersect a given simplex, we use a data structure for orthogonal range reporting, built in $\mathbb{R}^2$ on the points $P$. How we can report the pyramids using such a data structure is shown in Theorem 5.19. We first prove the following lemma.

**Lemma 5.18.** *Let $p \in \mathbb{R}^2$ be a point. A simplex $\Delta \subset \mathbb{R}^3$ lies above $\nabla_p$ if and only if all vertices of $\Delta$ lie above $\nabla_p$, and any unbounded edge of $\Delta$, when seen as a ray originating from some vertex of $\Delta$, has direction $\vec{d}$, with*

$$\begin{cases} -\vec{d}_z \leq \vec{d}_x \leq \vec{d}_z, \\ -\vec{d}_z \leq \vec{d}_y \leq \vec{d}_z. \end{cases}$$

**Proof.** Let $p \in \mathbb{R}^2$ be a point and let $\Delta \subset \mathbb{R}^3$ be a simplex. Because $\Delta$ is a convex object, and because the area above $\nabla_p$ forms a convex region, we have that $\Delta$ lies above $\nabla_p$ if and only if all vertices and edges of $\Delta$ lie above $\nabla_p$. By the same argument, an edge of $\Delta$ lies above $\nabla_p$ if and only if its incident vertices lie above $\nabla_p$. This proves the theorem for bounded simplices.

Assume that $\Delta$ is unbounded, and that all vertices of $\Delta$ lie above $\nabla_p$. Let $e$ be an unbounded edge of $\Delta$, incident to vertex $v$. Let $\vec{d}$ be the direction of $e$, when seen as a ray originating from $v$. A point $(x, y, z)$ on $e$ now satisfies

$$\begin{cases} x = v_x + \lambda \vec{d}_x \\ y = v_y + \lambda \vec{d}_y \\ z = v_z + \lambda \vec{d}_z \end{cases}$$

for some $\lambda \geq 0$.

A point $(x, y, z)$ lies above $\nabla_p$ if and only if $d_\infty(p, (x, y)) \leq z$. It follows that $e$ lies above $\nabla_p$ if and only if

$$\max\left\{|p_x - (v_x + \lambda \vec{d}_x)|, |p_y - (v_y + \lambda \vec{d}_y)|\right\} \leq v_z + \lambda \vec{d}_z$$

for all $\lambda \geq 0$, and therefore if and only if the system of equations

$$\begin{cases} v_x + v_z + \lambda(\vec{d}_x + \vec{d}_z) \geq p_x \\ v_x - v_z + \lambda(\vec{d}_x - \vec{d}_z) \leq p_x \\ v_y + v_z + \lambda(\vec{d}_y + \vec{d}_z) \geq p_y \\ v_y - v_z + \lambda(\vec{d}_y - \vec{d}_z) \leq p_y \end{cases}$$

is satisfied for all $\lambda \geq 0$. Using our assumption that all vertices of $\Delta$, and $v$ in particular, lie above $\nabla_p$, we now have that $e$ lies in $\nabla$ if and only if the system of equations

$$\begin{cases} \lambda(\vec{d}_x + \vec{d}_z) \geq 0 \\ \lambda(\vec{d}_x - \vec{d}_z) \leq 0 \\ \lambda(\vec{d}_y + \vec{d}_z) \geq 0 \\ \lambda(\vec{d}_y - \vec{d}_z) \leq 0 \end{cases}$$

is satisfied for all $\lambda \geq 0$. This implies that $e$ is contained in $\nabla_p$ if and only if $-\vec{d}_z \leq \vec{d}_x \leq \vec{d}_z$ and $-\vec{d}_z \leq \vec{d}_y \leq \vec{d}_z$. □

**Theorem 5.19.** *In $O(n \log n)$ time, we can build a data structure of $O(n)$ size, that can report the pyramids in $\nabla$ that intersect a given simplex, and which lie below a given query point inside the simplex, in $O(\sqrt{n} + k)$ time, where $k$ is the number of reported pyramids. Alternatively, with $O(n \log n)$ space, the query time can be reduced to $O(\log n + k)$.*

**Proof.** Let $\Delta \subset \mathbb{R}^3$ be a simplex and let $q \in \Delta$ be a point. Because $q \in \Delta$, a pyramid $\nabla_p \in \nabla$ intersects $\Delta$ and lies below $q$ if and only if it lies below $q$ and it does not contain $\Delta$. The pyramid $\nabla_p$ lies under $q$ if and only if $d_\infty(p, (q_x, q_y)) \leq q_z$. That is, $\nabla_p$ lies under $q$ if and only if $p \in [q_x - q_z, q_x + q_z] \times [q_y - q_z, q_y + q_z] = S_q$. See also Figure 13. Now, using Lemma 5.18, we can check in $O(1)$ time whether $\Delta$ does not lie above $\nabla_p$, or whether it lies above $\nabla_p$ if and only if the vertices of $\Delta$ lie above $\nabla_p$. Without loss of generality, assume that $\Delta$ lies above $\nabla_p$ if and only if all its vertices lie above $\nabla_p$, and assume that $\Delta$ is a bounded simplex.

Like for the query point $q$, a vertex $v$ of $\Delta$ lies above $\nabla_p$ if and only if $p$ lies in some two-dimensional axis-aligned rectangle $S_v$. Therefore, the points $p$ that we have to report are those in the region

$$R = S_q \setminus \left( \bigcap_{v \in V_\Delta} S_v \right),$$

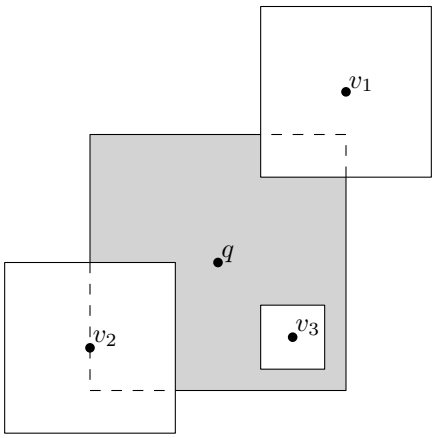



Figure 14: The region $R$ constructed in the proof of Theorem 5.19. The gray region is $R = S_q \setminus \left(\bigcap_{v \in V_\Delta} S_v\right)$.

where $V_\Delta$ is the set of vertices of $\Delta$. See Figure 14 for an illustration of this region. Because the intersection of two axis-aligned rectangles is an axis-aligned rectangle, the region $R$ is the difference of two axis-aligned rectangles. As there are only $O(1)$ vertices and edges in the union of the rectangles, their vertical decomposition will consist of $O(1)$ axis-aligned rectangles as well. By performing orthogonal range reporting on these rectangles, making sure not to include edges of the vertical decomposition in multiple queries, we can now report the points $p \in P$, and therefore the pyramids $\nabla_p$, for which $\nabla_p$ intersects $\Delta$ and lies below $q$, using $O(1)$ orthogonal range reporting queries. Using a $k$-d tree or range tree for these queries results in the bounds in the theorem. □

The number of pyramids intersecting a simplex $\Delta \in \Xi$ is $O(n/r)$. To see this, we only need to bound the number of pyramids that intersect the boundary of $\Delta$, but not its interior, as the number of pyramids intersecting the interior of $\Delta$ is at most $n/r$. Note that if a pyramid $\nabla_p$ intersects the boundary of $\Delta$ but not its interior, it must be that $\nabla_p$ contains either a vertex of $\Delta$, or the point $(p_x, p_y, 0)$ (the apex of $\nabla_p$) must lie on a face of $\Delta$.

Fix a vertex $v$ of $\Delta$. For any pyramid $\nabla_p$ containing $v$, we have have $\max\{|p_x - v_x|, |p_y - v_y|\} = v_z$. As this region is formed by four line segments, and because the points of $P$ lie in general position, it follows that at there are at most eight points $p \in P$ for which $\nabla_p$ contains $v$. Therefore, there are at most $O(1)$ pyramids that contain a vertex of $\Delta$.

Now fix a face $F$ of $\Delta$. If a point $p \in P$ lies on $F$, it must lie on the intersection between $F$ and the $xy$-plane. This intersection is a single line segment, from which it follows that at most two points of $P$ lie on $F$. Therefore, there are at most $O(1)$ points of $P$ that lie on a face of $\Delta$. It follows that the total number of pyramids that intersect $\Delta$ is $O(n/r)$. This leads to the following corollary.

**Corollary 5.20.** *In $O(n \log n)$ time, we can build a data structure of $O(n)$ size, that reports the pyramids of $\nabla$ that intersect a simplex $\Delta \in \Xi$, and which lie below a given point in $\Delta$, in $O(\sqrt{n} + n/r)$ time. Alternatively, with $O(n \log n)$ space, the query time can be reduced to $O(\log n + n/r)$.*

The final part of the data structure is the set of range-counting data structures for the different sets of pyramids with equal colors. For each color $c$, let $\nabla_c \subset \nabla$ be the set of pyramids with color $c$. Because a point $q$ lies above a pyramid $\nabla_p$ if and only if $d_\infty(p, (q_x, q_y)) \leq q_z$, counting the number of pyramids below $q$ that have color $c$ is equivalent to counting the number of points of $P$ in the range $S((q_x, q_y), q_z) = [q_x - q_z, q_x + q_z] \times [q_y - q_z, q_y + q_z]$ that have color $c$. The range-counting data structures therefore take the form of orthogonal range-counting data structures.

Let $P_c \subset P$ be the set of points that have color $c$. Using either a $k$-d tree or range tree, the total preprocessing time for these data structures is

$$O\left(\sum_c |P_c| \log |P_c|\right) = O(n \log n).$$

Using a $k$-d tree, the total space for the range-counting data structures is $O(n)$, and for a range tree, the total space will be $O(n \log n)$.

The time taken to answer a range mode query is $O(\log n)$ to perform a point-location query, resulting in a simplex $\Delta$ being reported that contains the query point. Reporting the pyramids intersecting $\Delta$ can then be done in $O(\sqrt{n} + n/r)$ time using $O(n)$ space, or $O(\log n + n/r)$ time using $O(n \log n)$ space, by Corollary 5.20. Let $C_\Delta$ be the set of different colors of the reported pyramids. The final step is to count the frequencies of these colors among the pyramids below the query point, as well as counting the frequency of the color stored in $\Delta$. Using a $k$-d tree, this step takes

$$O\left(\sum_{c \in C_\Delta} \sqrt{|P_c|} + \sqrt{n}\right)$$

time, which we can bound using Hölders inequality (see Section 5.3.2) as

$$O\left(\sqrt{n/r} \cdot \sqrt{\sum_{c=1}^{n/r} |P_c|}\right) = O(n/\sqrt{r}).$$

Using a range tree, the range counting queries take $O((n/r) \log n)$ time in total.

We summarize the result in the following theorem.

**Theorem 5.21.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$ and let $1 \leq r \leq n$. In $O(n \log n + nr^3)$ time, we can build a data structure of $O(n + r^3)$ size, that answers square range mode queries in $O(n/\sqrt{r})$ time. Alternatively, with $O(n \log n + r^3)$ space, the query time can be changed to $O((n/r) \log n)$.*

### 5.3.4 A better preprocessing algorithm

In the analyses of Sections 5.3.2 and 5.3.3, the dominating term for the preprocessing time is that of computing the colors stored in the simplices. This was done by simply scanning through the set of planes or pyramids to find the mode color among those below a given simplex, and repeating this procedure for all $O(r^3)$ simplices. The resulting procedure takes $O(nr^3)$ time. However, aside from this procedure, the preprocessing time is only $O(nr^2)$. In this section, we show how to lower the time taken to compute the colors to $O(n^{1+\delta} + n^{2/3} \cdot r^3 + nr^2)$ for the case of planes, and $O(n \log n + \sqrt{n} \cdot r^3 + nr^2)$ (using $O(n)$ space) or $O(n \log n + r^3 \log n + nr^2)$ (using $O(n \log n)$ space) for the case of pyramids. These complexities are better than $O(nr^3)$ for certain values of $r$, which include the values that we set $r$ to later on.

Let $S$ be a set of $n$ surfaces in $\mathbb{R}^3$, that are either all planes, or all upside-down pyramids. Let $\Xi$ be a $(1/r)$-cutting of $\mathcal{A}(S)$. The main idea behind the faster preprocessing algorithm is that we can look at the cutting $\Xi$ as a graph $G = (V, E)$, where $V$ is the set of vertices of $\Xi$ and $E$ is the set of bounded edges of $\Xi$. The algorithm traverses this graph using depth-first search, and keeps track of the frequencies of all colors among those surfaces of $S$ that lie strictly below the current vertex. This information can then be converted into the color that we want to store in an incident simplex, using the following observation.

**Observation 5.22.** Let $\Delta \in \Xi$ be a simplex and let $v$ be a vertex of $\Delta$. A surface lies strictly below $\Delta$ if and only if it lies strictly below $v$, and if it does not intersect $\Delta$.

For the algorithm, we use the following data structure, built on those surfaces of $S$ that lie strictly below the current vertex. For each color $c$, we store its frequency $f_c$ among the surfaces in the data structure. We also keep a set of $n + 1$ linked lists $L_0, \ldots, L_n$, such that list $L_f$ stores those colors whose frequency is $f$. As we want to quickly find the location of a color inside a linked list, we store pointers for each color, pointing to the element in a linked list containing the color. That is, for color $c$, we store a pointer to color $c$ in list $L_{f_c}$. See Figure 15 for an illustration of this part of the data structure. To keep track of the current highest frequency, we also keep the highest index $f^*$ for which $L_{f^*}$ is non-empty. This index will be used to quickly find a mode color, as every color stored in $L_{f^*}$ will have the highest frequency. Finally, we need a data structure that can report the surfaces that intersect a given simplex $\Delta \in \Xi$, and which lie below a query point $q \in \Delta$. In the case where $S$ consists of planes, this data structure is that of Corollary 5.15. When $S$

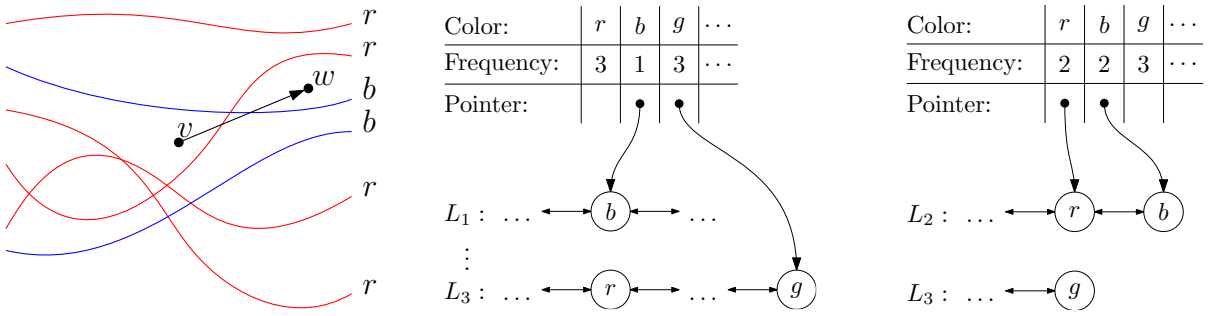



Figure 15: The data structure used during the alternative preprocessing algorithm. On the left is the situation at vertex $v$, which has three red surfaces and one blue surface below it. In the middle is the status of the data structure at vertex $v$. On the right is the status of the data structure after the algorithm moves from vertex $v$ to $w$, which has one less red surface and one more blue surface below it.

consists of pyramids, this data structure is that of Corollary 5.20. Let $P(n)$ be the preprocessing time of this data structure, and let $Q(n)$ be its query time (both depending on the kind of surfaces that are in $S$).

During traversal of $G$, we insert and delete surfaces from the data structure. This leads to us incrementing and decrementing the frequencies of the different colors. To increment the frequency of color $c$, we first find its current frequency, which is $f_c$. We can then find color $c$ inside list $L_{f_c}$ in $O(1)$ time, by following the pointer that we stored. Removing the color from the linked list can be done in $O(1)$ time, now that we have the location of the color. Then we insert color $c$ into list $L_{f_c+1}$ and adjust the stored pointer to point to the new location of color $c$. This can also be done in $O(1)$ time. Finally, it might be the case that $c$ was a mode color, which would result in the frequency $f^*$ being incremented. To account for this, we set $f^* \leftarrow \max\{f^*, f_c+1\}$. The whole incrementing procedure takes $O(1)$ time in total. Decrementing the frequency of a color works similarly, with the only real difference being how $f^*$ is updated. This is because $f^*$ must only be updated when $f^* = f_c$ and when $L_{f_c}$ is now empty. If this is the case, we simply set $f^* \leftarrow f_c - 1$. As we can check whether a list is empty in $O(1)$ time, decrementing a frequency takes $O(1)$ time as well. We can therefore insert and delete surfaces from the data structure in $O(1)$ time.

The initial data structure can be built as follows. Initially, every color has frequency 0. We therefore set every entry $f_c$ to 0, add every color to list $L_0$, and set $f^* \leftarrow 0$. Say we now start the traversal at vertex $v$. Using a linear scan over $S$, we can find those surfaces that lie strictly below $v$. Insert each of these surfaces into the data structure with the above procedure. This takes $O(n)$ time in total. The construction time of the data structure for reporting the surfaces intersecting a simplex $\Delta \in \Xi$, and which lie below a query point $q \in \Delta$, is $P(n)$, making the total initialization time $O(P(n) + n)$.

When traversing an edge $e = (v, w)$, we can update the data structure by removing those surfaces that lie strictly below $v$ and (not necessarily strictly) above $w$, and by inserting those surfaces that lie above $v$ and strictly below $w$. See Figure 15 for an illustration of how the data structure changes when traversing an edge. Note that the surfaces that have to be removed or inserted all intersect $\Delta$. Therefore, we can report a superset of these surfaces, containing $O(n/r)$ surfaces, in $Q(n)$ For each of the reported surfaces, we can check in $O(1)$ time whether it has to be removed or inserted (or ignored), after which we can update the data structure in $O(1)$ time. In total, we use $O(Q(n) + n/r)$ time to update the data structure when traversing an edge.

For every vertex $v$ encountered during the traversal, we check its incident simplices in $\Xi$ to see if they already have a color stored or not. If a simplex $\Delta$ does not have a color stored yet, we compute its color as follows. The current data structure contains all surfaces strictly below $v$. A surface strictly below $v$ lies strictly below $\Delta$ if and only if it does not intersect $\Delta$. Therefore, we report all surfaces that *do* intersect $\Delta$, and which lie below $v \in \Delta$, and remove those reported surfaces which strictly lie below $v$ from the data structure. Finding and removing these surfaces takes $O(Q(n) + n/r)$ time. The result is a data structure containing precisely the surfaces strictly below $\Delta$. Therefore, the frequency $f^*$ is the frequency of the color that we want to store in $\Delta$. Any color stored in list $L_{f^*}$ has this frequency, so we choose the head node of the list as the color to store in $\Delta$. For any vertex, we can thus compute the color to store in an incident simplex in $O(Q(n)+n/r)$ time in total. Afterwards, we need to undo the changes made to the data structure

by reinserting the surfaces that we deleted. For this, we can again report the surfaces below $v$ that intersect $\Delta$, and insert the surfaces that lie strictly below $v$. This step also takes $O(Q(n) + n/r)$ time.

Combining all these components, we have a graph traversal algorithm that takes $O(Q(n) + n/r)$ time to traverse an edge. Traversing all edges therefore takes $O(Q(n)r^3 + nr^2)$ time. The total time spent in the vertices checking incident simplices is $O(r^3)$, as each simplex can only be checked at most four times (once per vertex). The total time spent computing the colors of incident vertices is $O(Q(n)r^3 + nr^2)$, as it only has to be done once per simplex. The total time spent in the vertices is therefore $O(Q(n)r^3 + nr^2)$. The result is an $O(P(n) + Q(n)r^3 + nr^2)$ time preprocessing algorithm for the data structure from Section 5.3.1.

In the case where $S$ consists of planes, we have that $P(n) = O(n^{1+\delta})$ and $Q(n) = O(n^{2/3})$, by Corollary 5.15. Adapting Theorem 5.16 with this new preprocessing algorithm, we achieve the following result.

**Theorem 5.23.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$ and let $1 \leq r \leq n$. In $O(n^{1+\delta} + n^{2/3} \cdot r^3 + nr^2)$ time, we can build a data structure of $O(n + r^3)$ size, that answers halfspace range mode queries in $O(n/r^{1/3})$ time, where$\delta > 0$ is an arbitrarily small constant.*

By setting $r = n^{1/3}$ and combining the result with Theorem 5.9 (setting $r = n^{1/4}$), we get the following result for the chromatic $k$-nearest neighbors problem.

**Theorem 5.24.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$. In $O(n^{5/3})$ time, we can build a data structure of $O(n)$ size, that answers chromatic $k$-nearest neighbors queries on $P$ under the $L_2$ metric in $O(n^{8/9})$ time.*

In the case where $S$ consists of pyramids, we have that $P(n) = O(n \log n)$ and either that $Q(n) = O(\sqrt{n})$ (using $O(n)$ space) or $Q(n) = O(\log n)$ (using $O(n \log n)$ space), by Corollary 5.20. Adapting Theorem 5.21 with this new preprocessing algorithm, we achieve the following results.

**Theorem 5.25.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$ and let $1 \leq r \leq n$. In $O(n \log n + \sqrt{n} \cdot r^3 + nr^2)$ time, we can build a data structure of $O(n + r^3)$ size, that answers square range mode queries in $O(n/\sqrt{r})$ time. Alternatively, in $O(n \log n + r^3 \log n + nr^2)$ time, we can build a data structure of $O(n \log n + r^3)$ size, that answers queries in $O((n/r) \log n)$ time.*

By setting $r = n^{1/3}$ and combining the result with Theorem 5.12, we obtain the following result for the chromatic $k$-nearest neighbors problem.

**Theorem 5.26.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$. In $O(n^{5/3})$ time, we can build a data structure of $O(n)$ size, that answers chromatic $k$-nearest neighbors queries on $P$ under the $L_1$ and $L_\infty$ metrics in $O(n^{5/6})$ time. Alternatively, with $O(n \log n)$ space, the query time can be lowered to $O(n^{2/3} \log n)$ time.*

# 6 The two-dimensional semi-online problem

In this section we alter the data structures given in Section 5 to solve the semi-online problem. Recall the following definition for this problem.

| Semi-online two-dimensional chromatic $k$-nearest neighbors | |
|---|---|
| **Problem:** | Build a data structure supporting insertions and deletions of points, given that the time at which a point is deleted is known, such that given a query point $q \in \mathbb{R}^2$ and integer $k \in [1, n]$, the most frequent color among the $k$-nearest neighbors of $q$ can be found efficiently. |

First we make the data structures that report a range containing the $k$-nearest neighbors of a query point semi-online. This is done in Section 6.1 for the problem under the $L_2$ metric, and in Section 6.2 when the $L_1$ or $L_\infty$ metric is used. Then in Section 6.3, we present a semi-online version of the data structure in Section 5.3 for range mode queries.

## 6.1 Dynamically finding the $k$-nearest neighbors under the $L_2$ metric

First we make the data structure from Section 5.1 semi-online. For this, we again work in three-dimensional space, after applying the lifting map from Section 5.1.1. To keep notation simple, we let $P$ denote the three-dimensional lifted set of points. Say we have a static data structure built on $P$ and its set of dual planes $P^*$.

The static data structure consisted of a $(1/r)$-cutting of $\mathcal{A}(P^*)$, as well as the simplex range-searching data structure of Matoušek [44] discussed in Theorem 5.8 that can report or count the (number of) points of $P$ inside a convex query polyhedron. Given a two-dimensional query point $q$, the data structure from Section 5.1 returns the highest three-dimensional point in dual space, along a vertical line $\ell$ that depends on $q$, such that the returned point has exactly $k-1$ planes below it. The planes (not necessarily strictly) below this point are dual to the lifted $k$-nearest neighbors of $q$. See Figure 11 in Section 5.1.2.

The reason that we cannot make the data structure fully online is because of the range-searching data structure, and in particular the reporting queries. Both reporting and counting queries are are *decomposable*, meaning we can answer a query on disjoint subsets of $P$ that together make up $P$, and combine the results to find the result of querying on $P$. This makes the range-searching data structure suitable for applying the technique of Bentley and Saxe [14] to make it support insertions (see Section 6.2). The problem is that if we want to make it support deletions using the technique of Bentley and Saxe, the operation used to combine results of a decomposable query must have an inverse. This is the case for range counting (the operation is addition, its inverse is subtraction), but not for reporting (the operation is set-union). We therefore settle for making the data structure semi-online.

To make the data structure semi-online, we handle the updates in sets of $u$ consecutive updates each, called *rounds*. Before each round of updates, we split the points that are in the data structure into two sets $M$ and $D$. Set $D$ contains all points that will be deleted in the coming round, and set $M$ contains all other points. To know what points will be deleted in the coming round, we keep a red-black tree [24], which we call the *event queue*, that stores the points in the data structure sorted by the time that they are deleted, from lowest to highest. Aside from this event queue, we build the static data structure of Section 5.1 on $M$.

Insertions and deletions of points are handled using $D$, leaving $M$ the same during a round. At any time, the set $M \cup D$ will be the set of points currently in the data structure. When a point is inserted, the event queue gets updated with its deletion time. After a round, we move the points that are left in $D$ to $M$ (these points were inserted during the round and never deleted), and we make $D$ again contain those points that will be deleted in the coming round. These points are found using the event queue. The static data structure is then rebuilt on $M$.

Let $n$ be the maximum number of points in $M \cup D$ during a round. The space used is $O(n + r^3)$ for the data structure on $M$, where $1 \leq r \leq n$, and $O(u)$ for the set $D$, totalling $O(n + r^3)$ space. Inserting or deleting a point during a round can be done in $O(\log u)$ time, by keeping track of $D$ using for example a red-black tree [24]. Performing $u$ updates then takes $O(u \log u)$ time. Updating $M$ and $D$ between rounds can be done in $O(u \log n)$ time, as it takes $O(\log n)$ time to find a point in the event queue and to move it from $M$ to $D$. Rebuilding the static data structure takes $O(n^{1+\delta} + nr^2)$ time, or $O(n^{1+\delta} + r^5)$ time if $r \leq n^\alpha$ for some constant $\alpha < 1/3$, by Theorem 5.9. The total update time, when amortized over the $u$ updates, is therefore $O((n^{1+\delta} + nr^2)/u + \log u)$ or $O((n^{1+\delta} + r^5)/u + \log u)$ if $r \leq n^\alpha$ for some constant $\alpha < 1/3$.

Performing a query is done as follows. Let $\ell$ be the vertical query line along which we want to find the highest point $H_q^*$ whose level with respect to $M \cup D$ is equal to $k-1$. We first query the static data structure on $M$ twice, both times using $\ell$, but one time with the parameter $k-u$ and one time with the parameter $k$. The result is two points $h_-^*$ and $h_+^*$ on $\ell$, with $h_-^*$ being the highest point whose level in $\mathcal{A}(M)$ is equal to $k-u-1$, and with $h_+^*$ being the highest point whose level in $\mathcal{A}(M)$ is $k-1$. Adding in the points of $D$, we now have that the level of $h_-^*$ in $\mathcal{A}(M \cup D)$ is between $k-u-1$ and $k-1$, which means that $H_q^*$ lies above $h_-^*$. The level of $h_+^*$ in $\mathcal{A}(M \cup D)$ will be between $k-1$ and $k+u-1$, which means that $H_q^*$ lies below $h_+^*$. See Figure 16 for an illustration of the situation.

Because the difference in levels between $h_-^*$ and $h_+^*$ is at most $(k+u-1)-(k-u-1) = 2u$, there are at most $2u$ planes between $h_-^*$ and $h_+^*$. By Corollary 5.6, the point $H_q^*$ must lie on one of these planes. Using Lemma 5.3 and Theorem 5.8, these planes can be reported in $O(n^{2/3} + u)$ time with the data structure of Matoušek [44]. We can find out the exact level $k_+$ of $h_+^*$ in $\mathcal{A}(M \cup D)$ in $O(n^{2/3} + u)$ time, by range counting on $M$ and scanning over $D$ afterwards. The plane that contains $H_q^*$ is then the $(k^+ - k)$-th highest plane. We can find this plane by computing and sorting the intersection points between $\ell$ and the reported planes from highest to lowest, taking $O(u \log u)$ time.

Using the result of Theorem 5.9 for the static data structure, we summarize the result for the semi-online data structure in the following theorem.

**Theorem 6.1.** *Let $1 \leq r, u \leq n$. We can build a semi-online data structure of $O(n + r^3)$ size that, given a query point $q \in \mathbb{R}^2$, finds the smallest disk $D_q$ centered at $q$ which contains precisely the $k$-nearest neighbors*

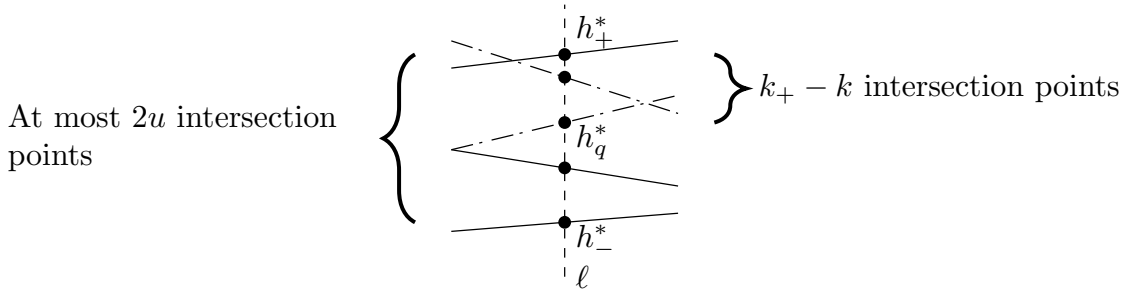



Figure 16: The situation between $h_+^*$ and $h_-^*$. Regular lines are in the set $M$, while dash-dotted lines are in $D$. The level of $h_+^*$ in $\mathcal{A}(M \cup D)$ is $k_+$.

*of $q$ among the points in the data structure, in*

$$O\left(r^3 \log r + n^{2/3} \log r + (n/r)\log(n/r) + u \log u\right)$$

*time. This data structure supports updates in $O((n^{1+\delta}+nr^2)/u+\log u)$ amortized time, reducing to $O((n^{1+\delta}+r^5)/u+\log u)$ when $r \leq n^\alpha$ for a constant $\alpha < 1/3$.*

## 6.2 Dynamically finding the $k$-nearest neighbors under the $L_1$ and $L_\infty$ metrics

We now make the data structure of Section 5.2 fully dynamic. As shown in Lemma 5.10, we only have to consider the $L_\infty$ metrics, as we can reduce the problem under the $L_1$ metric to that under the $L_\infty$ metric in $O(n)$ time.

The static data structure worked by performing binary search on the different $x$- and $y$-coordinates of the points in $P$. Given a query point $q$ and an $x$-coordinate $p_x$ of some point $p \in P$, the number of points in the square $S(q, |q_x - p_x|)$ would be counted using an orthogonal range counting data structure to see if the desired square is bigger or smaller than this square. The same would be done for the $y$-coordinates.

Making the sequences of $x$- and $y$-coordinates dynamic is simple. We store the coordinates in two red-black trees (one per set of coordinates). These trees are balanced binary-search trees that support insertions and deletions in $O(\log n)$ time each [24].

### 6.2.1 Insertion-only range counting

We now show how to make the range counting data structure support insertions. The technique used for this is due to Bentley and Saxe [14], and works for any decomposable problem. This includes the problem of range counting, which is decomposable using addition for combining results. Say we have $n$ points stored currently. The insertion-only data structure then consists of $\ell = \lfloor \log n \rfloor + 1$ levels $L_1, \ldots, L_\ell$, with level $L_i$ either being empty, or containing a static orthogonal range counting data structure built on precisely $2^{i-1}$ points.

Upon inserting a point into this data structure, the first empty level is located. Let this level be $L_k$. The total number of points stored in the levels before $L_k$ is

$$\sum_{i=1}^{k-1} 2^{i-1} = 2^{k-1} - 1,$$

which means that we can move the points in the levels before $L_k$ over to $L_k$ and add the newly inserted point, while maintaining the invariant that level $L_i$ is either empty or contains $2^{i-1}$ points. The levels before $L_k$ now contain points that are stored in $L_k$, so their data structure can be made empty. See Figure 17 for an illustration of this procedure.

Let $S(n)$ be the space used by the orthogonal range counting data structure. Because $S(n)$ must be $\Omega(n)$, we have that $S(n_1)+S(n_2) = O(S(n_1+n_2))$. Because each point is stored in exactly one level, it now follows that the total space used is $O(S(n))$. Performing a query is done by performing range counting in all $O(\log n)$

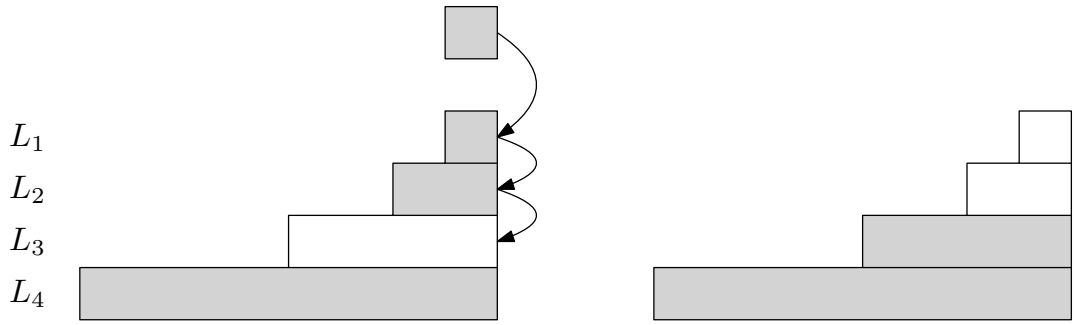



Figure 17: An illustration of how the insertion-only data structure handles insertions. White rectangles mark empty levels, gray rectangles mark full levels. On the left is the situation before inserting the new point, on the right is the situation after inserting the point.

levels and combining the results. In this way, queries can be performed in $O(Q(n) \log n)$ time, where $Q(n)$ is the query time of the range counting data structures. It remains to bound the update time of the data structure. For this, note that after removing a point from a level, it will never return to that level. A point will therefore only be in $O(\log n)$ different range counting data structures. Let $P(n)$ be the preprocessing time of the range counting data structures. Building level $L_i$ takes $P(2^{i-1})$ time, which we can amortize over the $2^{i-1}$ updates that have been performed to insert the points. The amortized update time is therefore

$$O\left(\sum_{i=1}^{O(\log n)} \frac{P(2^{i-1})}{2^{i-1}}\right) = O\left(P(n) \log n/n\right).$$

We summarize this in the following theorem.

**Theorem 6.2.** *Given a data structure of $S(n)$ size that can be build in $P(n)$ time, which answers orthogonal range counting queries in $Q(n)$ time, we can build a data structure of $O(S(n))$ size that answers orthogonal range counting queries in $O(Q(n) \log n)$ time, and which supports insertions in $O(P(n) \log n/n)$ amortized time.*

### 6.2.2 Fully dynamic range counting

We now extend the insertion-only data structure to also support deletions. We again make use of the technique of Bentley and Saxe [14], which works for any decomposable problem in which the operation used to combine results has an inverse. This is the case for range counting, as we combine results using addition, which has subtraction as its inverse.

To handle deletions, we keep two insertion-only data structures described above. The first is our main data structure $M$, and the second is a *ghost* data structure $G$. When a point is deleted, rather than deleting it from $M$, we insert it into $G$. Because these deleted points are still in $M$, we call them ghosts. The points in $M$ that are not ghosts will be called *alive*.

Queries are handled by querying $M$ and $G$ separately, and subtracting the result of $G$ from that of $M$. This does introduce the problem where too many points have been deleted, making $M$ and $G$ contain a lot more points than just the alive points. For this reason, we periodically rebuild $M$ on only those points that are not deleted, and reset $G$ to be empty. In particular, we rebuild after $G$ contains more than half of the points from $M$. Rebuilding the data structure takes

$$O\left(\sum_{i=1}^{O(\log n)} P(2^{i-1})\right) = O(P(n))$$

time, given that $P(n) = \Omega(n)$. Because we delete at least $n/2$ points before we have to rebuild the data structure again, and because deleting a point (inserting it into $G$) takes $O(P(n) \log n/n)$ amortized time, the amortized deletion time of the data structure is $O(P(n) \log n/n)$.

Insertions are handled by inserting the points into $M$, taking $O(P(n)\log n/n)$ time. A query is performed by querying both $M$ and $G$, and subtracting the result of $G$ from that of $M$. This is because $M$ returns the total number of points, alive or ghost, inside the data structure, while $G$ returns the number of ghosts alone. Queries are therefore handled in $O(Q(n)\log n)$ time, as before in the insertion-only case.

Using a $k$-d tree or range tree as the static orthogonal range counting data structure, we obtain the following result.

**Theorem 6.3.** *We can build a data structure of $O(n)$ size that answers orthogonal range counting queries in $O(\sqrt{n}\log n)$ time, and which supports insertions and deletions in $O(\log^2 n)$ amortized time. Alternatively, using $O(n\log n)$ space, the query time can be lowered to $O(\log^2 n)$.*

## 6.3 Semi-online range mode queries

We now make the data structure from Section 5.3 semi-online. Recall that for a set of $n$ surfaces $S$, this data structure consists of a $(1/r)$-cutting of $\mathcal{A}(S)$, together with a point-location data structure built on the cutting. A simplex in the cutting stores the most frequently occurring color among the surfaces strictly below the simplex. Aside from the cutting, the data structure consists of a data structure for counting the number of surfaces below a point, built on each set of surfaces $S_c$ with the same color. The final part of the data structure is a data structure that can report the surfaces intersecting a given simplex. A query works by finding the simplex in the cutting that contains the query point, reporting the surfaces intersecting that simplex, and performing range counting on the different colors among the reported surfaces, as well as on the color stored in the simplex. The color with the highest count occurs the most frequently below the query point.

We choose to make this data structure semi-online, rather than fully dynamic, because of the colors stored in the simplices of the cutting. When a surface is deleted, it can invalidate the colors of every simplex in the cutting, as any other color can now become the mode color strictly below a simplex. This can be countered by storing the $u$ most frequently occurring colors, but this increases the space used by the data structure with a factor $O(u)$, and will increase the query times, as there will be more colors to count the surfaces for. This tactic has not led to any meaningful complexities, and a different approach will be needed to obtain a fully dynamic data structure. As we have not been able to find one, we settle for finding a semi-online data structure.

To make the data structure for range mode queries semi-online, we apply the same tactic as in Section 6.1. For this, we again handle the updates in rounds of $u$ updates each. Before a round, we split up the set of surfaces into two sets $M$ and $D$, such that $D$ contains all surfaces that will be deleted in the coming round. However, as we will see later, it is important that for every color $c$, all surfaces with color $c$ are in the same set. Thus, if a surface of color $c$ is going to be deleted in the coming round, we add all (potentially $O(n)$) surfaces of color $c$ to $D$. To be able to quickly find all surfaces of a given color, we keep a red-black tree for each color, storing the surfaces of that color that are in the data structure. Using these trees, we can find all surfaces that need to be put in $D$ in $O(n)$ time.

Let $n$ be the maximum number of surfaces in the data structure during a round. We build the static data structure of Section 5.3 on $M$, and build a dynamic data structure for counting the number of surfaces below a point on each of the colors in $D$. Because range counting is decomposable using addition for combining results, and because addition has an inverse, we can use the techniques of Bentley and Saxe [14] to obtain a dynamic data structure for counting surfaces below a point.

Insertions and deletions are now handled directly on $D$, by inserting or deleting a surface from the correct data structure, which depends on the color of the surface. A query is handled as follows. Let $q$ be the query point. Because no color has a surface in both $M$ and $D$, we can decompose the query over $M$ and $D$. Once we have found the mode color among the surfaces in $M$ and among those in $D$, we can count their frequencies below $q$ to find the one that occurs the most frequently, which will be the mode of $M \cup D$ under $q$. Note that we can only do this because we made sure not to have surfaces of the same color in both sets.

**Analysis under the $L_2$ metric.** The static data structure for $M$ can be built in $O(n^{1+\delta} + n^{2/3}\cdot r^3 + nr^2)$ time and uses $O(n+r^3)$ space, by Theorem 5.23. By making the data structure of Matoušek [44] fully dynamic using the techniques of Bentley and Saxe [14], we obtain a fully dynamic data structure for halfspace range counting, and through duality, for counting the number of planes below a query point. On $n$ planes, this

data structure uses $O(n)$ space, answers queries in $O(n^{2/3} \log n)$ time, and supports updates in $O(n^{\delta} \log n)$ amortized time. Amortizing the time taken to build the data structure for $M$ over the $u$ updates in a round, we now obtain a total amortized update time of

$$O((n^{1+\delta} + n^{2/3} \cdot r^3 + nr^2)/u).$$

To find the mode color below $q$ in $M$, we perform a query on the static data structure, taking $O(n/r^{1/3})$ time by Theorem 5.23. To find the mode color below $q$ in $D$, we perform a range counting query for every color in $D$. Let $D_c$ be the set of planes in $D$ with color $c$. The counting queries take

$$O\left(\sum_c |D_c|^{2/3} \log |D_c|\right) = O\left(\sum_c |D_c|^{2/3} \log n\right)$$

time in total, which we can bound using Hölders inequality (see Section 5.3.2). Because the number of colors in $D$ is at most $u$, Hölders inequality lets us bound the total time for the range-counting queries as

$$O\left(u^a \cdot \left[\sum_{c=1}^{u} |D_c|^{a+b}\right]^b \log n\right)$$

for all positive values $a$ and $b$. By setting $a = 1/3$ and $b = 2/3$, it follows that the total time taken for the queries is $O(u^{1/3} \cdot n^{2/3} \log n)$. The total query time of the semi-online data structure for halfspace range mode is therefore $O(n/r^{1/3} + u^{1/3} \cdot n^{2/3} \log n)$.

We summarize the result in the following theorem.

**Theorem 6.4.** *Let $1 \leq r, u \leq n$. We can build a semi-online data structure of $O(n + r^3)$ size that answers halfspace range mode queries in $O(n/r^{1/3} + u^{1/3} \cdot n^{2/3} \log n)$ time, and which supports updates in $O((n^{1+\delta} + n^{2/3} \cdot r^3 + nr^2)/u)$ amortized time.*

By setting $r = n^{3/10}$ and $u = n^{7/10}$, we can combine the result with that of Theorem 6.1 (setting $r = n^{1/4}$ and $u = n^{3/4}$) to achieve the following result.

**Theorem 6.5.** *We can build a semi-online data structure of $O(n)$ size that answers chromatic $k$-nearest neighbors queries under the $L_2$ metric in $O(n^{9/10} \log n)$ time, and which supports updates in $O(n^{9/10})$ amortized time.*

**Analysis under the $L_1$ and $L_\infty$ metrics.** We first analyze the data structure when only linear space is available. The static data structure for $M$ can be built in $O(n \log n + \sqrt{n} \cdot r^3 + nr^2)$ time and uses $O(n + r^3)$ space, by Theorem 5.25. By making a $k$-d tree fully dynamic using the techniques of Bentley and Saxe [14], we obtain a fully dynamic data structure for orthogonal range counting, which can be used to count the number of upside-down pyramids (graphs of distance functions under the $L_\infty$ metric) below a query point. On $n$ pyramids, this data structure uses $O(n)$ space, answers queries in $O(\sqrt{n} \log n)$ time, and supports updates in $O(\log^2 n)$ amortized time. Amortizing the time taken to build the data structure for $M$ over the $u$ updates in a round, we now obtain a total amortized update time of

$$O((n \log n + \sqrt{n} \cdot r^3 + nr^2)/u).$$

To find the mode color below $q$ in $M$, we perform a query on the static data structure, taking $O(n/\sqrt{r})$ time by Theorem 5.25. To find the mode color below $q$ in $D$, we perform a range counting query for every color in $D$. Let $D_c$ be the set of pyramids in $D$ with color $c$. The counting queries take

$$O\left(\sum_c \sqrt{|D_c|} \log |D_c|\right) = O\left(\sum_c \sqrt{|D_c|} \log n\right)$$

time in total, which we can bound using Hölders inequality (see Section 5.3.2). Because the number of colors in $D$ is at most $u$, Hölders inequality lets us bound the total time for the range-counting queries as

$$O\left(u^a \cdot \left[\sum_{c=1}^{u} |D_c|^{a+b}\right]^b \log n\right)$$

for all positive values $a$ and $b$. By setting $a = 1/2$ and $b = 1/2$, it follows that the total time taken for the queries is $O(\sqrt{u \cdot n} \log n)$. The total query time of the semi-online data structure for square range mode is therefore $O(n/\sqrt{r} + \sqrt{u \cdot n} \log n)$.

We summarize the result in the following theorem.

**Theorem 6.6.** *Let $1 \leq r, u \leq n$. We can build a semi-online data structure of $O(n + r^3)$ size that answers square range mode queries in $O(n/\sqrt{r} + \sqrt{u \cdot n} \log n)$ time, and which supports updates in $O((n \log n + \sqrt{n} \cdot r^3 + nr^2)/u)$ amortized time.*

By setting $r = n^{2/7}$ and $u = n^{5/7}$, we can combine the result with that of Theorem 6.3 to achieve the following result.

**Theorem 6.7.** *We can build a semi-online data structure of $O(n)$ size that answers chromatic $k$-nearest neighbors queries under the $L_1$ and $L_\infty$ metrics in $O(n^{6/7} \log n)$ time, and which supports updates in $O(n^{6/7})$ amortized time.*

When $O(n \log n)$ space is available, we can use the result of Theorem 5.25 to create a static data structure on $M$ that uses $O(n \log n + r^3)$ space, which can be built in $O(n \log n + \sqrt{n} \cdot r^3 + nr^2)$ time, and which answers queries in $O((n/r) \log n)$ time. Instead of using a dynamic $k$-d tree to count the pyramids below a point, we can make a range tree dynamic, again using the techniques of Bentley and Saxe [14]. On $n$ pyramids, the resulting data structure uses $O(n \log n)$ space, answers queries in $O(\log^2 n)$ time, and supports updates in $O(\log^2 n)$ amortized time. Amortizing the time taken to build the data structure for $M$ over the $u$ updates in a round, we now obtain a total amortized update time of

$$O((n \log n + \sqrt{n} \cdot r^3 + nr^2)/u).$$

To find the mode color below $q$ in $M$, we perform a query on the static data structure, taking $O((n/r) \log n)$ time by Theorem 5.25. To find the mode color below $q$ in $D$, we perform a range counting query for every color in $D$. As there are at most $u$ different colors in $D$, these range counting queries take $O(u \log^2 n)$ time in total. The total query time of the semi-online data structure for square range mode is therefore $O((n/r) \log n + u \log^2 n)$.

We summarize the result in the following theorem.

**Theorem 6.8.** *Let $1 \leq r, u \leq n$. We can build a semi-online data structure of $O(n \log n + r^3)$ size that answers square range mode queries in $O((n/r) \log n + u \log^2 n)$ time, and which supports updates in $O((n \log n + \sqrt{n} \cdot r^3 + nr^2)/u)$ amortized time.*

By setting $r = n^{1/4}$ and $u = n^{3/4}$, we can combine the result with that of Theorem 6.3 to achieve the following result.

**Theorem 6.9.** *We can build a semi-online data structure of $O(n)$ size that answers chromatic $k$-nearest neighbors queries under the $L_1$ and $L_\infty$ metrics in $O(n^{3/4} \log^2 n)$ time, and which supports updates in $O(n^{3/4})$ amortized time.*

# 7 The two-dimensional approximate problem

In this section we give data structures for the approximate problems, which were defined as follows.

| TWO-DIMENSIONAL APPROXIMATE CHROMATIC $k$-NEAREST NEIGHBORS | |
|---|---|
| **Given:** | A set of $n$ colored points $P \subset \mathbb{R}^2$ in general position, and an approximation factor $\varepsilon \in (0, 1)$. |
| **Problem:** | Preprocess $P$ into a data structure, such that given a query point $q \in \mathbb{R}^2$ and integer $k \in [1, n]$, a color occurring at least $(1-\varepsilon)$ times as often as the most frequent color among the $k$-nearest neighbors of $q$ can be found efficiently. |

To solve the approximate problems, we will not rely on the tactic of making separate approximate versions for finding the $k$-nearest neighbors and performing range mode queries. The reason is this. Let $m$ be the

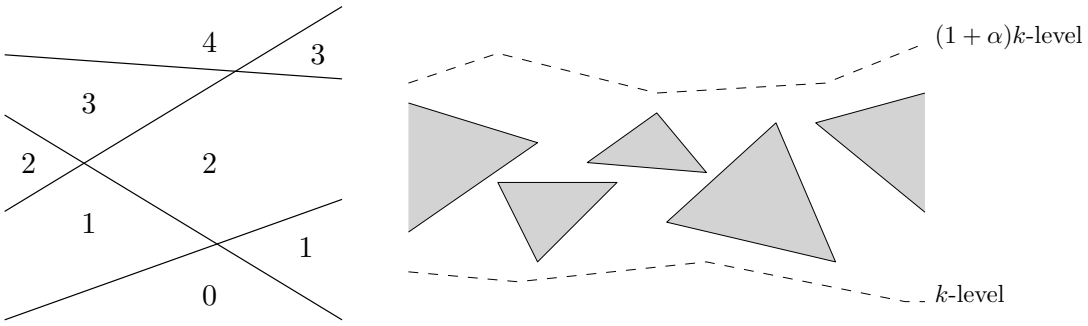



Figure 18: An illustration of (approximate) levels in arrangements. On the left is an arrangement of lines, where each region is marked with the level of the points inside. On the right is an $\alpha$-approximate $k$-level.

mode color among the $k$-nearest neighbors of a point, and let $f_m$ be its frequency. Say we have now found a range containing only $(1-\varepsilon)k$ of the $k$-nearest neighbors of a point. Then it can happen that the frequency of $m$ inside this range is only $f_m - \varepsilon k$. If we then look for an approximate mode in this range, we could end up reporting a color with a frequency of $(1-\varepsilon)\cdot(f_m - \varepsilon k) < (1-\varepsilon)f_m$. The reported color would then not be a valid approximate mode for the $k$-nearest neighbors. Luckily though, by solving the approximate range mode problem, where we wish to report a color that occurs often enough inside a given range, we can solve the chromatic $k$-nearest neighbors problem as well.

## 7.1 Approximate mode queries

Recall from Section 5.1 that when the $L_2$ metric is used, we can transform $P$ to a set of planes in $\mathbb{R}^3$ such that chromatic $k$-nearest neighbors queries can be answered by finding the mode color among the $k$-lowest planes in $\mathbb{R}^3$ along a vertical query line. Here we define the $k$-lowest surfaces along a vertical line $\ell$ as the set of surfaces for which at most $k-1$ surfaces intersect $\ell$ strictly below them. Also, recall from Section 5.3.3 that under the $L_1$ and $L_\infty$ metrics, the problem is equivalent to that of finding the mode color among the $k$-lowest upside-down pyramids in $\nabla = \{\nabla_p \mid p \in P\}$ along a vertical query line in $\mathbb{R}^3$, where an upside-down pyramid $\nabla_p$ was defined as the graph of the distance function $d_\infty(p,q)$ for a point $p$.

Let $S$ be a set of $n$ planes (in the case of the $L_2$ metric) or upside-pyramids (in the case of the $L_1$ and $L_\infty$ metrics). We first focus on the problem where, given a query point $q$, we wish to report a color that occurs at least $(1-\varepsilon)$ times as often as the mode color among those surfaces below $q$. After we have a data structure for this problem, we show how to extend it to handle approximate chromatic $k$-nearest neighbors queries.

Our solution is based on the data structures for halfspace range counting by Afshani and Chan [1] and Har-Peled et al. [35]. The data structure consists of multiple *approximate levels* for the arrangement $\mathcal{A}(S)$. The *$k$-level* of $\mathcal{A}(S)$, for $1 \le k \le n$, is the set of points that lie on a surface in $S$ and which have exactly $k$ surfaces passing strictly below them. We define an *$\alpha$-approximate $k$-level* of $\mathcal{A}(S)$ as a set of simplices, such that:

1. Each simplex lies between the $k$-level and $(1+\alpha)k$-level of $\mathcal{A}(S)$.
2. Every vertical line intersects at least one simplex in the approximate level.

See Figure 18 for an illustration of (approximate) levels in an arrangement of lines. The concepts are similar for general surfaces. The reason for approximating the $k$-level is that the $k$-level in an arrangement of planes, for example, is $O(nk^{3/2})$ [48], which is too high when we want to achieve (near-)linear space. Approximate levels have much lower complexities.

The work of Kaplan et al. [37] shows how to compute such an approximate level, as Kaplan et al. construct an approximate level while constructing a *vertical shallow cutting* of the *$(\le k)$-level* of $\mathcal{A}(S)$. The $(\le k)$-level of $\mathcal{A}(S)$ is the set of points that have at most $k$ surfaces of $S$ passing strictly below them. A vertical shallow cutting of the $(\le k)$-level is a collection of vertical semi-unbounded *prisms* (the set of points below some triangle), whose union contains the $(\le k)$-level, and in which the prisms intersect at most $O(k)$ surfaces of $S$. See Figure 19 for an illustration of approximate levels and vertical shallow cuttings.

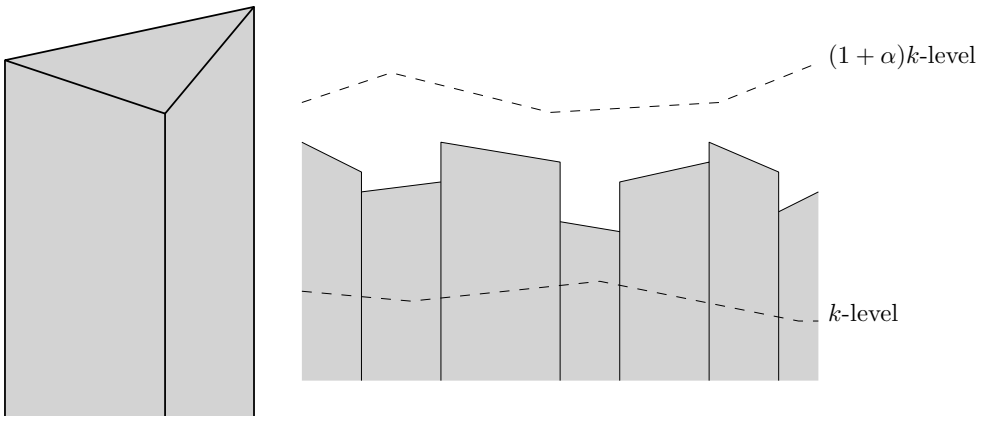



Figure 19: On the left is a vertical semi-unbounded prism. On the right is a vertical shallow cutting for the $(\leq k)$-level, built with vertical semi-unbounded prisms, in which the top triangles form an $\alpha$-approximate $k$-level.

Kaplan et al. [37] specifically construct a terrain between the $k$-level and $(1+\alpha)k$-level of an arrangement of certain types of surfaces, with the additional property that the prisms induced by the triangles in the terrain form a shallow cutting for the $(\leq k)$-level. As such a terrain is an $\alpha$-approximate $k$-level, we can use the work of Kaplan et al. to construct the approximate levels. The result is the following theorem. Note that the result uses the function $\lambda_s(n)$, which is the maximum length of a Davenport-Schinzel sequence of order $s$ on $n$ symbols, where $s$ depends on the type of surfaces that the cutting is created on. In the case of $S$, consisting of planes or upside-down pyramids, we have $s = 4$ [37]. As shown by Agarwal and Sharir [7], $\lambda_4(n) = O(n \cdot 2^{\alpha(n)})$, where $\alpha(n)$ is the inverse-Ackermann function.

**Theorem 7.1.** *Let $1 \leq m \leq n$ and let $S_m \subseteq S$ be a subset consisting of $m$ surfaces of $S$. Let $1 \leq k \leq m$ and let $\alpha \in (0, 1/2]$. In*

$$O\left(\frac{m\log^3 m}{\alpha^2} \cdot \lambda_4\left(\frac{\log m}{\alpha^2}\right)\right)$$

*expected time, we can build an $\alpha$-approximate $k$-level of $\mathcal{A}(S_m)$ with*

$$O\left(\frac{m\log^2 m}{\alpha^5 k}\right)$$

*deterministic complexity.*

**Proof.** We summarize Theorem 8.1 of Kaplan et al. [37], which deals with constructing the approximate terrain. Note that while Kaplan et al. construct the terrain for $\alpha = 1/2$, we generalize the proof to work for $\alpha \in (0, 1/2]$.

Let $S_k \subseteq S_m$ be a random sample of size $\min\left\{\frac{cm}{\alpha^2 k}\log m, m\right\}$, where $c > 0$ is a suitable constant. Pick $t$ uniformly at random from the range

$$\left[\left(1+\frac{\alpha}{3}\right)\lambda, \left(1+\frac{\alpha}{2}\right)\lambda\right],$$

where $\lambda = \frac{c}{\alpha^2}\log m$. Kaplan et al. prove that the $t$-level of $\mathcal{A}(S_k)$ is a terrain lying between the $k$-level and $(1+\alpha)k$-level of $\mathcal{A}(S_m)$ with high probability (with probability at least $1 - \frac{1}{m^b}$, where $b \geq 1$ depends on $c$). The expected complexity of this level is

$$O\left(\frac{m\log^2 m}{\alpha^5 k}\right),$$

and it can be constructed in

$$O\left(mt\lambda_4(t)\log\left(\frac{m}{t}\right)\log m\right) = O\left(\frac{m\log^3 m}{\alpha^2} \cdot \lambda_s\left(\frac{\log m}{\alpha^2}\right)\right)$$

expected time. By constructing a constant number of these levels in expectation, the complexity of the resulting level can be made deterministic.

Let $\overline{T}_k$ be the $t$-level of $\mathcal{A}(S_k)$. To check whether $\overline{T}_k$ is really a terrain between the $k$-level and $(1+\alpha)k$-level, we try to create a vertical shallow cutting from it, in which each prism intersects at most $(1+\alpha)k$ surfaces of $S_m$. To do this, we create the vertical decomposition $\overline{\Lambda}_k$ of $\overline{T}_k$. This takes $O((m/k)\log^3 m)$ deterministic time. We now have that each prism in $\overline{\Lambda}_k$ intersects at most $(1+\alpha)k$ surfaces of $S_m$ with high probability, and if this is the case, then $\overline{T}_k$ is a terrain between the $k$-level and $(1+\alpha)k$-level of $\mathcal{A}(S_m)$. Checking whether this is the case takes

$$O\left(\frac{m\log^3 m}{\alpha^5}\right)$$

time in expectation.

By performing the above procedure a constant number of times in expectation, a terrain $\overline{T}_k$ can be found that lies between the $k$-level and $(1+\alpha)k$-level of $\mathcal{A}(S_m)$. As such a terrain is an $\alpha$-approximate $k$-level of $\mathcal{A}(S_m)$, the theorem is proven. □

As an $\alpha$-approximate $k$-level is also an $(\alpha+\beta)$-approximate $k$-level for all $\beta > 0$, we can construct $1/2$-approximate $k$-levels instead of $\alpha$-approximate $k$-levels when $\alpha > 1/2$. This results in the following corollary.

**Corollary 7.2.** *Let $1 \leq m \leq n$ and let $S_m \subseteq S$ be a subset consisting of $m$ surfaces of $S$. Let $1 \leq k \leq m$ and let $\alpha > 0$. Let $\beta = \min\{\alpha, 1/2\}$. In*

$$O\left(\frac{m\log^3 m}{\beta^2}\cdot\lambda_4\left(\frac{\log m}{\beta^2}\right)\right)$$

*expected time, we can build an $\alpha$-approximate $k$-level of $\mathcal{A}(S_m)$ with*

$$O\left(\frac{m\log^2 m}{\beta^5 k}\right)$$

*deterministic complexity.*

We are now ready to give the data structure. Set $\alpha = 1-\sqrt{1-\varepsilon}$ and $\beta = \min\left\{\frac{\alpha}{1-\alpha}, \frac{1}{2}\right\}$. Let $S_c$ be the set of surfaces in $S$ with color $c$. For each set $S_c$, we build $\left(\frac{\alpha}{1-\alpha}\right)$-approximate $\left(\frac{1}{1-\alpha}\right)^i$-levels $L_{c,i}$ of $\mathcal{A}(S_c)$, for $i = 0, \ldots, \log_{\frac{1}{1-\alpha}} |S_c|$. Note that for any $c$ and $i$, the simplices in $L_{c,i}$ lie between the levels $\left(\frac{1}{1-\alpha}\right)^i$ and $\left(\frac{1}{1-\alpha}\right)^{i+1}$ of $\mathcal{A}(S_c)$. The main idea behind the data structure now comes from the following theorem.

**Theorem 7.3.** *Let $q \in \mathbb{R}^3$ be a point. Let $f_m$ be the exact frequency of the mode color among the surfaces below $q$ and let $i$ be the biggest integer for which there is a color $c$ such that $q$ lies above $L_{c,i}$. Then any color $c$ for which $q$ lies above the approximate level $L_{c,i}$ occurs at least $(1-\varepsilon)f_m$ times among the surfaces below $q$.*

**Proof.** Let $c$ be a color for which $q$ lies above the approximate level $L_{c,i}$, and therefore above the $\left(\frac{1}{1-\alpha}\right)^i$-level of $\mathcal{A}(S_c)$. Then the number of surfaces below $q$ that have color $c$ is at least $\left(\frac{1}{1-\alpha}\right)^i + 1 = \left(\frac{1}{1-\varepsilon}\right)^{i/2} + 1$. Because $i$ is the biggest integer for which $q$ lies above $L_{c,i}$, we have that $q$ lies strictly below $L_{c,i+1}$, and therefore strictly below the $\left(\frac{1}{1-\alpha}\right)^{i+2}$-level of $\mathcal{A}(S_c)$. The number of surfaces below $q$ that have color $c$ is therefore strictly lower than $\left(\frac{1}{1-\alpha}\right)^{i+2} + 1 = \left(\frac{1}{1-\varepsilon}\right)^{i/2+1} + 1$. Let $f_c$ be the number of surfaces below $q$ that have color $c$. Then we now have that

$$\left(\frac{1}{1-\varepsilon}\right)^{i/2} + 1 \leq f_c < \left(\frac{1}{1-\varepsilon}\right)^{i/2+1} + 1,$$

from which it follows that

$$(1-\varepsilon)f_c < \left(\frac{1}{1-\varepsilon}\right)^{i/2} + 1 \leq f_c.$$

The above inequality holds in particular for the mode color.

Let $f_m$ be the frequency of the mode color below $q$. Then for any color $c$ for which $q$ lies above $L_{c,i}$, with $f_c$ its frequency, we have that

$$(1-\varepsilon)f_m < \left(\frac{1}{1-\varepsilon}\right)^{i/2} + 1 \leq f_c,$$

which proves the theorem. □

Given an integer $i$, we can check whether there is a color $c$ for which $q$ lies above $L_{c,i}$ by looking at the *lower envelope* of $\bigcup_c L_{c,i}$. The lower envelope $\mathrm{LE}(i)$ of the approximate levels $L_{c,i}$ has the property that a point lies above $\mathrm{LE}(i)$ if and only if there is a color $c$ for which the point lies above $L_{c,i}$. Unfortunately, while for a given $i$ the total complexity of the approximate levels $L_{c,i}$ is

$$O\left(\sum_c \frac{|S_c|\log^2|S_c|}{\beta^5 \cdot \left(\frac{1}{1-\alpha}\right)^i}\right) = O\left(\frac{n\log^2 n}{\beta^5 \cdot \left(\frac{1}{1-\alpha}\right)^j}\right) = O\left(\frac{n\log^2 n}{\beta^5}\right),$$

the total complexity of $\mathrm{LE}(i)$ will be more than quadratic in $n$ [28, 46]. We therefore turn to the problem of *ray shooting among triangles*.

Checking whether $q$ lies above $\mathrm{LE}(i)$ can be done by shooting a vertical ray upwards from the point $(q_x, q_y, -\infty)$, into the set of triangles formed by the faces of simplices in the levels $L_{c,i}$. The intersection between the ray and the first triangle hit by it will lie in $\mathrm{LE}(i)$, and therefore, we have that $q$ lies above $\mathrm{LE}(i)$ if and only if the reported triangle lies below $q$. Because $\mathrm{LE}(i+1)$ lies completely above $\mathrm{LE}(i)$, we can combine binary search with this ray-shooting procedure to find the highest integer $i$ such that $q$ lies above $\mathrm{LE}(i)$. By coloring each triangular face of a simplex in a level $L_{c,i}$ with color $c$, the reported triangle in the highest lower envelope below $q$ will be an approximate mode, using Theorem 7.3.

The problem of finding the first triangle hit by a given ray is the problem of ray shooting among triangles. See Figure 20 for an illustration of ray shooting. Data structures for this problem on $n$ triangles exist that use $O(n^{1+\delta})$ space, can be build in $O(n^{1+\delta})$ time, and that answer queries in $O(n^{3/4+\delta})$ time [6]. By using the fact that we only ever shoot vertical rays upwards, we can use a data structure for shooting rays in a fixed direction. Such a data structure exists that has the same space and preprocessing time complexities as the more general structure, but its query time is reduced to $O(n^{2/3+\delta})$ [25]. If we also use the fact that we only shoot rays from points whose $z$-coordinate is $-\infty$, we can reduce the query time even further, to $O(n^{1/2+\delta})$, while also reducing the space to $O(n)$. This is shown in the following lemma.

**Lemma 7.4.** *Let $S$ be a set of $n$ (possibly intersecting) triangles in $\mathbb{R}^3$. In $O(n^{1+\delta})$ time, we can build a data structure of $O(n)$ size, that can report the first triangle hit by a vertical ray shot upwards from a point $(q_x, q_y, -\infty)$ in $O(n^{1/2+\delta})$ time, where $\delta > 0$ is an arbitrarily small constant.*

**Proof.** We modify the data structure of de Berg [25] for ray shooting from a fixed point to work for our problem. Let $\rho$ be a vertical ray shot upwards from a point $(q_x, q_y, -\infty)$, and let $q = (q_x, q_y)$. Consider a triangle $t \in S$, and let $\bar{t}$ be the projection of $t$ onto the $xy$-plane. Then $t$ is hit by $\rho$ if and only if $q \in \bar{t}$. Without loss of generality, assume that $t$ and $\bar{t}$ are bounded. Let $\ell_1(\bar{t})$, $\ell_2(\bar{t})$ and $\ell_3(\bar{t})$ be the three lines through the edges of $\bar{t}$, in no particular order, and let $\ell_i(\bar{t})^+$ be the halfplane bounded by $\ell_i(\bar{t})$ that contains $\bar{t}$. Then $q \in \bar{t}$ if and only if $q \in \ell_i(\bar{t})^+$ for all $i$. This property will be used to find all triangles hit by $\rho$.

The data structure is a three-layered *partition tree*. We refer to [25] for a description of these multi-layer partition trees. At layer $i$, where $i = 1, 2, 3$, the lines $\ell_i(\bar{t})^+$ are stored, for all projected triangles $\bar{t}$. The leaves in the third layer correspond to the triangles of $S$. A node in the third layer corresponds to the set of triangles stored in the leaves below it. This set is called the *canonical subset* of the node. Using the partition tree, we can find the unique set of $O(n^{1/2+\delta})$ disjoint canonical subsets that contain precisely the projected triangles $\bar{t}$ that contain $q$, and therefore the triangles $t \in S$ intersected by $\rho$.

Let $S(\rho)$ be the set of triangles intersected by $\rho$, and let $H_{S(\rho)}$ be the set of planes containing the triangles in $S(\rho)$. Note that we now have that a triangle $t \in S(\rho)$ is the first triangle hit by $\rho$ if and only if the plane

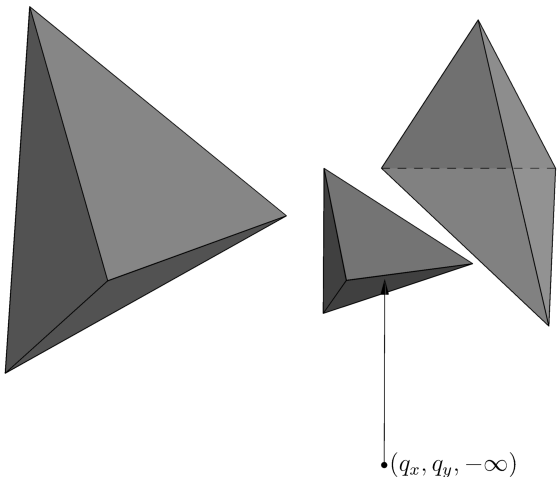



Figure 20: Shooting a vertical ray upwards from $(q_x, q_y, -\infty)$, into a set of simplices.

containing $t$ is the first plane in $H_{S(\rho)}$ hit by $\rho$. Because $\rho$ is a vertical ray shot upwards from $(q_x, q_y, -\infty)$, the first plane in $H_{S(\rho)}$ to be hit by $\rho$ can be found by planar point location in the $xy$-projection of the lower envelope of $H_{S(\rho)}$. These point-location data structures can be build during preprocessing, in each node of the third layer of the partition tree. Note that while the lower envelope of a set of triangles can have quadratic complexity [28, 46], the lower envelope of a set of planes only has linear complexity. This is why we first reduce the problem to ray shooting among planes.

For the proofs of the complexities of the partition tree, see [25]. The space used by the partition tree, including the point-location data structures at the third layer, is $O(n)$. Finding the canonical subsets that make up $S(\rho)$ can be done in $O(n^{1/2+\delta})$ time, resulting in $O(n^{1/2+\delta})$ canonical subsets. Performing point location in each canonical subset takes $O(\log n)$ time, after which combining the results to find the first triangle hit takes $O(n^{1/2+\delta})$ time. The query time is therefore $O(n^{1/2+\delta} \log n) = O(n^{1/2+\delta})$. The preprocessing time of the data structure, including the point-location data structures, is $O(n^{1+\delta})$. This follows from [25], as de Berg creates a similar data structure to ours, substituting every point-location data structure with the data structure of Agarwal and Matoušek [5] for ray shooting in a convex polytope. On a canonical subset of size $m$, this data structure can be constructed in $O(m^{1+\delta})$ time. As a point-location data structure on the same set takes only $O(m \log m)$ time to construct, the construction time of our entire data structure will not be higher than that of the data structure of de Berg. This proves the lemma. □

We now analyze the complexities of the entire approximation data structure. The total complexity of the approximate levels $L_{c,i}$, and therefore the space used by the ray shooting data structures, is

$$O\left(\sum_c \sum_{i=1}^{\log_{\frac{1}{1-\alpha}} |S_c|} \frac{|S_c| \log^2 |S_c|}{\beta^5 \cdot \left(\frac{1}{1-\alpha}\right)^i}\right) = O\left(\frac{n \log^2 n}{\beta^5} \cdot \sum_{i=1}^{\log_{\frac{1}{1-\alpha}} n} \frac{1}{\left(\frac{1}{1-\alpha}\right)^i}\right) = O\left(\frac{n \log^2 n}{\alpha \cdot \beta^5}\right).$$

The time taken to build the approximate levels is

$$\begin{aligned} O\left(\sum_c \sum_{i=1}^{\log_{\frac{1}{1-\alpha}} |S_c|} \frac{|S_c| \log^3 |S_c|}{\beta^2} \cdot \lambda_4\left(\frac{\log |S_c|}{\beta^2}\right)\right) &= O\left(\sum_c \frac{|S_c| \log^3 |S_c|}{\beta^2} \cdot \lambda_4\left(\frac{\log n}{\beta^2}\right) \log_{\frac{1}{1-\alpha}} n\right) \\ &= O\left(\frac{n \log^3 n}{\beta^2} \cdot \lambda_4\left(\frac{\log n}{\beta^2}\right) \log_{\frac{1}{1-\alpha}} n\right). \end{aligned}$$

in expectation. It follows from Lemma 7.4 and the total complexity of the approximate levels that the time taken to build the ray shooting data structures is (deterministic) $O((n \log^2 n/(\alpha \cdot \beta^5))^{1+\delta})$. The query time for a ray shooting query is $O((n \log^2 n/\beta^5)^{1/2+\delta})$, as for a single $i$, the approximate levels $L_{c,i}$ had a total

complexity of $O(n \log^2 n/\beta^5)$. By performing binary search over the integers $1 \leq i \leq \log_{\frac{1}{1-\alpha}} n$, querying the ray-shooting data structure at each step, we can now find a color that occurs at least $(1-\varepsilon)$ times as frequent as the mode color below a query point in

$$O\left(\left(\frac{n \log^2 n}{\beta^5}\right)^{1/2+\delta} \log\log_{\frac{1}{1-\alpha}} n\right)$$

deterministic time.

Note that $\varepsilon/2 \leq \alpha = 1 - \sqrt{1-\varepsilon} \leq \varepsilon$ and that $\beta = \min\left\{\frac{\alpha}{1-\alpha}, \frac{1}{2}\right\} \geq \min\{\alpha, 1/2\}$, so $\alpha = \Theta(\varepsilon)$ and $\beta = \Omega(\min\{\varepsilon, 1/2\}) = \Omega(\varepsilon)$. This will let us rewrite the bounds in terms of $\varepsilon$. We summarize the above results in the following theorem.

**Theorem 7.5.** *Let $\varepsilon \in (0,1)$. In*

$$O\left(\frac{n \log^3 n}{\varepsilon^2} \cdot \lambda_4\left(\frac{\log n}{\varepsilon^2}\right) \log_{\frac{1}{1-\varepsilon}} n + \left(\frac{n \log^2 n}{\varepsilon^6}\right)^{1+\delta}\right)$$

*expected time, we can build a data structure of $O\left(\frac{n \log^2 n}{\varepsilon^6}\right)$ deterministic size, that given a query point reports a color occurring at least $(1-\varepsilon)$ times as often as the mode color among the surfaces of $S$ below the query point in*

$$O\left(\left(\frac{n \log^2 n}{\varepsilon^5}\right)^{1/2+\delta} \log\log_{\frac{1}{1-\varepsilon}} n\right)$$

*deterministic time, where $\delta > 0$ is an arbitrarily small constant.*

We can extend the above data structure to support approximate chromatic $k$-nearest neighbors queries in the same asymptotic bounds. Recall that this problem is equivalent to finding a color that occurs at least $(1-\varepsilon)$ times as often among the $k$-lowest planes along a query line as the most often occurring color. For this equivalent problem, we build, besides the data structure described above, a data structure for counting the number of surfaces of $S$ below a given query point. Under the $L_2$ metric, we will use the data structure of Matoušek [44]. Under the $L_1$ and $L_\infty$ metrics, we will use a $k$-d tree. These data structures are chosen as their complexities will not dominate in the final data structure.

Let $\ell$ be a vertical query line, going through the point $(\ell_x, \ell_y, 0)$. The procedure still consists of a binary search over the integers $1 \leq i \leq n$. At every step, we perform ray shooting with a vertical ray shooting upwards from the point $(\ell_x, \ell_y, -\infty)$, into the set of triangles formed by the faces of simplices in the levels $L_{c,i}$. The intersection between the ray and the reported triangle will lie in $\mathrm{LE}(i)$.

Say we have now reported a triangle $\Delta$, whose intersection with line $\ell$ is $\Delta_\ell$. Its level can be computed by counting the number of surfaces strictly below it. Using this combination of ray shooting and counting surfaces, we can find the highest $i$ such that the level of the reported triangle $\Delta_\ell$ is at most $k-1$. The highest point on $\ell$ that has a level of at most $k-1$ lies above the reported triangle. Therefore, by Theorem 7.3, the color of $\Delta_\ell$ will be an approximate mode for the $k$-lowest surfaces along $\ell$.

This results in the following theorem.

**Theorem 7.6.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$ and let $\varepsilon \in (0,1)$. In*

$$O\left(\frac{n \log^3 n}{\varepsilon^2} \cdot \lambda_4\left(\frac{\log n}{\varepsilon^2}\right) \log_{\frac{1}{1-\varepsilon}} n + \left(\frac{n \log^2 n}{\varepsilon^6}\right)^{1+\delta}\right)$$

*expected time, we can build a data structure of $O\left(\frac{n \log^2 n}{\varepsilon^6}\right)$ deterministic size, that answers approximate chromatic $k$-nearest neighbors queries on $P$ under the $L_1$, $L_2$ and $L_\infty$ metrics in*

$$O\left(\left(\frac{n \log^2 n}{\varepsilon^5}\right)^{1/2+\delta} \log\log_{\frac{1}{1-\varepsilon}} n\right)$$

*deterministic time, where $\delta > 0$ is an arbitrarily small constant.*

## 7.2 Reducing the space and query time complexities

Agarwal et al. [3] prove the existence of a $(1/r)$-cutting of $S$ that covers only the $(\leq k)$-level of $\mathcal{A}(S)$, rather than all of $\mathbb{R}^3$, that has a size of $O(q^{2+\delta} \cdot r)$, where $1 \leq k, r \leq n$ and $q = k(r/n) + 1$. Setting $r = 2n/(\alpha k)$ proves the existence of a set of simplices $\Xi$ that cover the $(\leq (1+\alpha/2)k)$-level of $\mathcal{A}(S)$, which consists of $O\left(\frac{n}{\alpha^{3+\delta}k} + \frac{n}{\alpha k}\right)$ simplices, such that each simplex intersects at most $\alpha k/2$ surfaces of $\mathcal{A}(S)$. Note that a subset of this cutting will form an $\alpha$-approximate $k$-level of $\mathcal{A}(S)$. This shows the existence of an $\left(\frac{\alpha}{1-\alpha}\right)$-approximate $k$-level of size $O\left(\frac{n}{\alpha^{3+\delta}k} + \frac{n}{\alpha k}\right) = O_\varepsilon(n/k)$. Using these approximate levels in our data structure results in the following theorem.

**Theorem 7.7.** *Let $P$ be a set of $n$ points in $\mathbb{R}^2$ and let $\varepsilon \in (0,1)$. There exists a data structure of $O_\varepsilon(n)$ size, that answers approximate chromatic $k$-nearest neighbors queries under the $L_1$, $L_2$ and $L_\infty$ metrics in $O_\varepsilon(n^{1/2+\delta})$ time, where $\delta > 0$ is an arbitrarily small constant.*

When the $L_2$ metric is used, we can use the algorithm of Har-Peled et al. [35] for constructing approximate levels of $O_\varepsilon(n/k)$ size. For a given arrangement of $n$ planes in $\mathbb{R}^3$, their algorithm constructs a terrain that lies between the $k$-level and $(1+\alpha)k$-level of the arrangement, for any $1 \leq k \leq n$ and $\alpha > 0$. We can therefore directly use their algorithm for constructing the terrains $L_{i,j}$. The construction of an $\left(\frac{\alpha}{1-\alpha}\right)$-approximate $k$-level for $\mathcal{A}(S)$ takes

$$O\left(n + \frac{n \log^3 n}{\alpha^6 k}\right) = O\left(n + \frac{n \log^3 n}{\varepsilon^6 k}\right)$$

time in expectation, and results in a terrain with a deterministic complexity of $O\left(\frac{n}{\alpha^3 k}\right) = O\left(\frac{n}{\varepsilon^3 k}\right)$.

To construct all approximate levels used in our data structure, we now spend

$$O\left(\sum_c \left[|S_c| + \sum_{i=1}^{\log_{\frac{1}{1-\varepsilon}} |S_c|} \frac{|S_c| \log^3 |S_c|}{\varepsilon^6 \cdot \left(\frac{1}{1-\varepsilon}\right)^i}\right]\right) = O\left(n + \frac{n \log^3 n}{\varepsilon^6} \sum_{i=1}^{\log_{\frac{1}{1-\varepsilon}} n} \frac{1}{\left(\frac{1}{1-\varepsilon}\right)^i}\right) = O\left(\frac{n \log^3 n}{\varepsilon^7}\right)$$

time in expectation (note that we do not have to sum over the linear term in the construction of the levels when the color is the same, as it is noted by Har-Peled et al. [35] that this term comes from sampling the given set of points, which can be done for multiple levels simultaneously if the points stay the same). The total deterministic complexity of the approximate levels combined is

$$O\left(\sum_c \sum_{i=1}^{\log_{\frac{1}{1-\varepsilon}} |S_c|} \frac{|S_c|}{\varepsilon^3 \cdot \left(\frac{1}{1-\varepsilon}\right)^i}\right) = O\left(\frac{n}{\varepsilon^3} \cdot \sum_{i=1}^{\log_{\frac{1}{1-\varepsilon}} n} \frac{1}{\left(\frac{1}{1-\varepsilon}\right)^i}\right) = O(n/\varepsilon^4)$$

Because the complexities of the approximate levels are lower, the preprocessing, space and query time complexities of the ray shooting data structure are lowered as well.

We summarize the new complexities of the complete data structure in the following theorem.

**Theorem 7.8.** *Let $P \subset \mathbb{R}^3$ be a set of $n$ points and let $\varepsilon \in (0,1)$. In*

$$O\left(\frac{n \log^3 n}{\varepsilon^7} + \left(\frac{n}{\varepsilon^4}\right)^{1+\delta}\right)$$

*expected time, we can build a data structure of $O(n/\varepsilon^4)$ size, that answers approximate chromatic $k$-nearest neighbors queries on $P$ under the $L_2$ metric in*

$$O\left(\left(\frac{n}{\varepsilon^3}\right)^{1/2+\delta} \log\log_{\frac{1}{1-\varepsilon}} n\right)$$

*deterministic time, where $\delta > 0$ is an arbitrarily small constant.*

# 8 Conclusions & Future work

In this thesis, we gave solutions for a variety of problems based on the chromatic $k$-nearest neighbors problem. Mount et al. [45] first introduced this problem, and presented data structures whose query times depend linearly on $k$. To quote Mount et al.: "it is not clear how to determine the most common color among the $k$ nearest neighbors without explicitly computing the $k$ nearest neighbors."[3] We have found a solution to this problem, by working with a range that is guaranteed to contain only the $k$-nearest neighbors, and then computing the mode color inside this range. This has lead to a set of data structures whose query-time complexities do not depend on $k$, allowing for truly sublinear query times. We presented solutions for the regular problem, semi-online problem, and an approximate variant of the problem, under the $L_1$, $L_2$ and $L_\infty$ metrics.

While our achieved query times are sublinear, one can argue that they are on the high end. For the regular chromatic $k$-nearest neighbors problem, the achieved complexities come from those of the respective data structure for range mode queries, with the query times for finding a correct range being significantly lower. Like Chan et al. [20], we wonder if better results for range mode queries can be achieved, which likely requires entirely different approaches to the problem.

Turning to approximations unsurprisingly yielded better complexities, especially when the $L_2$ metric is used. Our formulation of the approximate problem comes from a version of approximate range counting. Perhaps different formulations, like that of Arya and Mount [8] for approximate range searching, or that of Arya et al. [9] for the approximate $k$-nearest neighbors problem, would lead to better results.

Our choices of metrics were the $L_1$, $L_2$ and $L_\infty$ metrics. These metrics were chosen because their unit circles have shapes that allowed us to create the data structures for finding a range containing the $k$-nearest neighbors and finding the mode color in this range. We believe that our data structures can be used for different metrics as well, as the main parts of the data structures have already been generalized to other metrics. In particular, $(1/r)$-cuttings have been generalized by Agarwal et al. [3] (through without efficient construction algorithms) and approximate levels by Kaplan et al. [37]. The data structures for range searching in certain types of ranges (orthogonal and halfspace ranges) can be replaced by data structures for semi-algebraic range searching (see for example [4, 6]). Whether our solutions can be generalized completely to other metrics, and what the complexities will be, remains open.

For this thesis, we only looked at the one- and two-dimensional problems. However, the data structures for the two-dimensional exact problems can all be generalized to higher dimensions, using higher dimensional versions of the results used for them. We wonder whether our solutions for the approximate problems can be generalized to higher dimensions, or whether different approaches are required. This is due to our approach requiring higher dimensional approximate levels (or higher dimensional shallow cuttings), the existence of which we are unaware of.

Finally, rather than looking at the dynamic problems, we studied the semi-online versions of the problems. This is mainly due to the range mode data structures, which we were not able to make fully dynamic. Finding fully dynamic data structures remains to be done, and we think it will require a different approach to either range mode queries, or the chromatic $k$-nearest neighbors problems in their entirety.

[3] Note that we talked about the *$k$-nearest neighbors* rather than the *$k$ nearest neighbors*.

# References


[1] P. Afshani and T. M. Chan, *On approximate range counting and depth*, Discrete & Computational Geometry, 42 (2009), pp. 3–21.

[2] P. Afshani and T. M. Chan, *Optimal halfspace range reporting in three dimensions*, in Proceedings of the Twentieth Annual ACM-SIAM Symposium on Discrete Algorithms, 2009, pp. 180–186.

[3] P. K. Agarwal, A. Efrat, and M. Sharir, *Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications*, SIAM Journal on Computing, 29 (1999), pp. 912–953.

[4] P. K. Agarwal, J. Matousek, and M. Sharir, *On range searching with semialgebraic sets II*, SIAM Journal on Computing, 42 (2013), pp. 2039–2062.

[5] P. K. Agarwal and J. Matoušek, *Ray shooting and parametric search*, in Proceedings of the 24th Annual ACM Symposium on Theory of Computing, S. R. Kosaraju, M. Fellows, A. Wigderson, and J. A. Ellis, editors, 1992, pp. 517–526.

[6] P. K. Agarwal and J. Matoušek, *On range searching with semialgebraic sets*, Discrete & Computational Geometry, 11 (1994), pp. 393–418.

[7] P. K. Agarwal and M. Sharir, *Davenport-Schinzel sequences and their geometric applications*, in Handbook of Computational Geometry, J. Sack and J. Urrutia, editors, 2000, pp. 1–47.

[8] S. Arya and D. M. Mount, *Approximate range searching*, Computational Geometry, 17 (2000), pp. 135–152.

[9] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, *An optimal algorithm for approximate nearest neighbor searching in fixed dimensions*, Journal of the ACM, 45 (1998), pp. 891–923.

[10] N. Bansal and R. Williams, *Regularity lemmas and combinatorial algorithms*, in 50th Annual IEEE Symposium on Foundations of Computer Science, 2009, pp. 745–754.

[11] A. Ben-Dor, L. Bruhn, N. Friedman, I. Nachman, M. Schummer, and Z. Yakhini, *Tissue classification with gene expression profiles*, Journal of Computational Biology, 7 (2000), pp. 559–583.

[12] J. L. Bentley, *Multidimensional binary search trees used for associative searching*, Communications of the ACM, 18 (1975), pp. 509–517.

[13] J. L. Bentley, *Multidimensional divide-and-conquer*, Communications of the ACM, 23 (1980), pp. 214–229.

[14] J. L. Bentley and J. B. Saxe, *Decomposable searching problems I. static-to-dynamic transformation*, Journal of Algorithms, 1 (1980), pp. 301–358.

[15] C. Bohler, P. Cheilaris, R. Klein, C. Liu, E. Papadopoulou, and M. Zavershynskyi, *On the complexity of higher order abstract voronoi diagrams*, in Automata, Languages, and Programming, F. V. Fomin, R. Freivalds, M. Z. Kwiatkowska, and D. Peleg, editors, 2013, pp. 208–219.

[16] P. Bose, E. Kranakis, P. Morin, and Y. Tang, *Approximate range mode and range median queries*, in STACS, V. Diekert and B. Durand, editors, 2005, pp. 377–388.

[17] T. M. Chan, *Semi-online maintenance of geometric optima and measures*, SIAM Journal on Computing, 32 (2003), pp. 700–716.

[18] T. M. Chan, *Optimal partition trees*, Discrete & Computational Geometry, 47 (2012), pp. 661–690.

[19] T. M. Chan, *Speeding up the four russians algorithm by about one more logarithmic factor*, in Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, P. Indyk, editor, 2015, pp. 212–217.

[20] T. M. Chan, S. Durocher, K. G. Larsen, J. Morrison, and B. T. Wilkinson, *Linear-space data structures for range mode query in arrays*, Theory of Computing Systems, 55 (2014), pp. 719–741.

[21] B. Chazelle, *An optimal convex hull algorithm and new results on cuttings (extended abstract)*, in Proceedings 32nd Annual Symposium on Foundations of Computer Science, 1991, pp. 29–38.

[22] B. Chazelle, *Cutting hyperplanes for divide-and-conquer*, Discrete & Computational Geometry, 9 (1993), pp. 145–158.

[23] B. Chazelle and J. Friedman, *A deterministic view of random sampling and its use in geometry*, in 29th Annual Symposium on Foundations of Computer Science, 1988, pp. 539–549.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 3rd Edition*, 2009.

[25] M. de Berg, *Ray Shooting, Depth Orders and Hidden Surface Removal*, Lecture Notes in Computer Science, 1993.

[26] D. P. Dobkin and S. Suri, *Maintenance of geometric extrema*, Journal of the ACM, 38 (1991), pp. 275–298.

[27] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, *Making data structures persistent*, Journal of Computer and System Sciences, 38 (1989), pp. 86–124.

[28] H. Edelsbrunner, *The upper envelope of piecewise linear functions: Tight bounds on the number of faces*, Discrete & Computational Geometry, 4 (1989), pp. 337–343.

[29] H. Edelsbrunner and E. P. Mücke, *Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms*, ACM Transactions on Graphics, 9 (1990), pp. 66–104.

[30] H. El-Zein, M. He, J. I. Munro, and B. Sandlund, *Improved time and space bounds for dynamic range mode*, in 26th Annual European Symposium on Algorithms, Y. Azar, H. Bast, and G. Herman, editors, 2018, pp. 25:1–25:13.

[31] I. Z. Emiris and J. F. Canny, *A general approach to removing degeneracies*, SIAM Journal on Computing, 24 (1995), pp. 650–664.

[32] M. J. Fischer and A. R. Meyer, *Boolean matrix multiplication and transitive closure*, in 12th Annual Symposium on Switching and Automata Theory, 1971, pp. 129–131.

[33] J. H. Friedman, J. L. Bentley, and R. A. Finkel, *An algorithm for finding best matches in logarithmic expected time*, ACM Transactions on Mathematical Software, 3 (1977), pp. 209–226.

[34] M. Greve, A. G. Jørgensen, K. D. Larsen, and J. Truelsen, *Cell probe lower bounds and approximations for range mode*, in Automata, Languages and Programming, S. Abramsky, C. Gavoille, C. Kirchner, F. M. auf der Heide, and P. G. Spirakis, editors, 2010, pp. 605–616.

[35] S. Har-Peled, H. Kaplan, and M. Sharir, *Approximating the k-level in three-dimensional plane arrangements*, in Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, R. Krauthgamer, editor, 2016, pp. 1193–1212.

[36] P. Horton and K. Nakai, *Better prediction of protein cellular localization sites with the k nearest neighbors classifier*, in Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology, T. Gaasterland, P. D. Karp, K. Karplus, C. A. Ouzounis, C. Sander, and A. Valencia, editors, 1997, pp. 147–152.

[37] H. Kaplan, W. Mulzer, L. Roditty, P. Seiferth, and M. Sharir, *Dynamic planar voronoi diagrams for general distance functions and their algorithmic applications*, Discrete & Computational Geometry, 64 (2020), pp. 838–904.

[38] D. Krizanc, P. Morin, and M. H. M. Smid, *Range mode and range median queries on lists and trees*, Nordic Journal of Computing, 12 (2005), pp. 1–17.

[39] D. T. Lee, *On k-nearest neighbor voronoi diagrams in the plane*, IEEE Transactions on Computing, 31 (1982), pp. 478–487.

[40] Y. Liao and V. R. Vemuri, *Using text categorization techniques for intrusion detection*, in Proceedings of the 11th USENIX Security Symposium, D. Boneh, editor, 2002, pp. 51–59.

[41] C. Liu, *Nearly optimal planar k nearest neighbors queries under general distance functions*, in Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, S. Chawla, editor, 2020, pp. 2842–2859.

[42] C. Liu, E. Papadopoulou, and D. T. Lee, *An output-sensitive approach for the $L_1$ / $L_\infty$ k-nearest-neighbor voronoi diagram*, in Proceedings of the 19th European Conference on Algorithms, C. Demetrescu and M. M. Halldórsson, editors, 2011, pp. 70–81.

[43] J. Matoušek, *Efficient partition trees*, in Proceedings of the Seventh Annual Symposium on Computational Geometry, R. L. S. Drysdale, editor, 1991, pp. 1–9.

[44] J. Matoušek, *Range searching with efficient hierarchical cuttings*, in Proceedings of the Eighth Annual Symposium on Computational Geometry, D. Avis, editor, 1992, pp. 276–285.

[45] D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, *Chromatic nearest neighbor searching: A query sensitive approach*, Computational Geometry, 17 (2000), pp. 97–119.

[46] J. Pach and M. Sharir, *The upper envelope of piecewise linear functions and the boundary of a region enclosed by convex plates: Combinatorial analysis*, Discrete & Computational Geometry, 4 (1989), pp. 291–309.

[47] M. I. Shamos and D. Hoey, *Closest-point problems*, in 16th Annual Symposium on Foundations of Computer Science, 1975, pp. 151–162.

[48] M. Sharir, S. Smorodinsky, and G. Tardos, *An improved bound for* k*-sets in three dimensions*, Discrete & Computational Geometry, 26 (2001), pp. 195–204.

[49] D. D. Sleator, R. E. Tarjan, and W. P. Thurston, *Rotation distance, triangulations, and hyperbolic geometry*, in Proceedings of the 18th Annual ACM Symposium on Theory of Computing, J. Hartmanis, editor, 1986, pp. 122–135.

[50] J. Snoeyink, *Point location*, in Handbook of Discrete and Computational Geometry, Second Edition, J. E. Goodman and J. O'Rourke, editors, 2004, pp. 767–785.

[51] R. F. Sproull, *Refinements to nearest-neighbor searching in k-dimensional trees*, Algorithmica, 6 (1991), pp. 579–589.

[52] D. E. Willard, *New data structures for orthogonal range queries*, SIAM Journal on Computing, 14 (1985), pp. 232–253.

[53] H. Yu, *An improved combinatorial algorithm for boolean matrix multiplication*, in Automata, Languages, and Programming, M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, 2015, pp. 1094–1105.