Computational Analysis of a Novel Chromatic k-Nearest Neighbours Algorithm

Thomas van der Plas, Frank Staals, Erwin Glazenburg

10 November 2024

Abstract

Classification algorithms are used in order to make predicitions based on historically available data. Chromatic k-Nearest Neighbour algorithms are an example of this, in which the the color of a query point is determined by the mode color of its k closest neighbours in historical data. Classical approaches to solving this problem have a runtime which is dependent on k or the amount of unique colors c. We investigate the computational performance of an alternative solution method proposed by van der Horst et al. in 2022 in which runtime is only dependent on the number of points in the historical dataset n. This should yield better performance when k or c are close to n.
We consider both range and mode finding in 1D and 2D contexts with a L_∞ distance metric, comparing to a naive k dependent implementation. We found that significant performance gains can be achieved from $k \gtrsim 300$ in 1D and $k \gtrsim 0.01n - 0.1n$ in 2D, validated against both artificial and real life data points.

1 Introduction

Classification algorithms are widely used in order to make predictions about new data points based on historical data. In general, a query point q is compared to a collection P of n existing data points with known classes. The class of q is then determined by considering the classes of points in P which are similar to q. Different algorithms apply different similarity metrics and techniques in order to determine what class is the most likely fit for our query.

One widely used example of a classification algorithm is that of k-Nearest Neighbours (k-NN). In this, the k points which are closest to q are considered. From this set, the mode (or most frequently occuring) class is selected as a prediction for the class of q. When modeling this as a geometric problem, generic class labels are often replaced with colors. Additionally, data points are represented as points in d-dimensional space. This version is generally known as a Chromatic k-Nearest Neighbours algorithm.

We are currently aware of only two results on the theory of chromatic k-NN algorithms. The first of these two is attributed to by Mount et al. [Mou+00]. They introduced an approach using linear space datastructures that are able to answer theories efficiently if colors within P are packed close together. For this, they devised a metric which they refer to as the chromatic density ρ which allowed them to get a query time of $O(\log^2 n + (1/\rho)^d \log(1/\rho))$.
The second result is attributed to van der Horst et al. [HLS22]. Their goal was to introduce a set of algorithms that with near-linear space usage could answer queries with a running time only dependent on n. They note that this would be especially advantagous in the event that either k or the amount of unique colors in P are close to n.
Our goal for this research is to get a better understanding of the advantages of having query times that are only dependent on n. We do this by implementing a selection of the algorithms presented in van der Horst et al. and comparing these with implementations of naive approaches.

Van der Horst et al. proposed a selection of different algorithms. They provide exact approaches for solving in 1, 2, or d dimensions with L_2, L_∞ and L_m distance metrics. They additionally provide approximation algorithms for $d = 1$ and $d = 2$ with a decreased runtime. In this research we will limit ourselves

to implementing the exact L_∞ approaches in both 1D and 2D. This selection was made as it provides a good base going forward. The 1D case provides us with the smallest possible instance of this problem, in which points are simply represented on a number line. The 2D case on the other hand forms the basis of the exact d-dimensional approach that was proposed, thereby allowing it to serve as a test of the real world applicability of the algorithm as a whole.

Each of the algorithms proposed by van der Horst et al. is split into two major steps. The first of these is range finding, in which the kth closest point from q is determined. This allows us to determine a radius from q in which all points lie that need to be considered. After this we apply mode finding, in which given q and our previously determined radius, we determine the mode color of the k closest points to q. This second step allows us to answer our query.
Following this two step split, we attempt to implement 4 seperate algorithms: 1D range finding, 1D mode finding, 2D range finding and 2D mode finding. Of these, we implement the first three as described by the original paper. 2D mode finding on the other hand could not directly be implemented as originally described; the preprocessing of this algorithm requires arrangements of planes to be built in 3D, however as far as we are aware major geometric computation libraries currently do not support this feature. As a result, we alter the original algorithm in order to only require arrangements in 2D. The underlying mechanisms of the algorithm are preserved, however due to this adjustment we can no longer answer mode queries for a certain radius. Instead, a halfplane mode query can be answered. We note that once 3D arrangements become available in geometric computation libraries, the groundwork that we laid down now should be easily adaptable allowing for radius based mode queries. We will therefore analyse the runtime of the implemented algorithm just as with the other 3 that were implemented without alteration.

The structure of this paper will be as follows. For all sections except the conclusion, we will seperate the 1D and 2D implementations. In this, we will split further by discussing range and mode finding operations seperately.
In Section 2, we will provide a summary of all the algorithms that were implemented. This also includes the naive approaches that were included as benchmarks for our results. In Section 3, we will discuss implementation details for each of the algorithms derived from van der Horst et al. Here we also discuss any possible modifications that were made in order to make algorithms computationally feasible. In Section 4, we discuss the test instances that we used and the results that our implementation produced. Then finally in Section 5, we will conclude, reflect and provide an overview of possible future work.

2 Methods

Here, we provide an overview of the computational complexities and outlines of the algorithms implemented in this paper. Only brief summaries of the vital parts of algorithms are provided; for more details, we refer the reader to the original work or to the source code in which the algorithms are implemented. An overview of all runtimes and space usage of the algorithms can be found in Table 1.

Algorithm	Preprocessing time	Space	Query time
1D L_m Naive range	$O(n \log n)$	$O(n)$	$O(\log n + k)$
1D L_m Fast range	$O(n \log n)$	$O(n)$	$O(\log n)$
1D L_m Naive mode	$O(n \log n)$	$O(n)$	$O(\log n + k)$
1D L_m Fast mode	$O(n^{3/2})$	$O(n)$	$O(\sqrt{n})$
2D L_∞ Naive range	-	$O(n)$	$O(n \log n)$
2D L_∞ Fast range	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n)$
2D L_∞ Naive mode	$O(n \log n)$	$O(n)$	$O(\log n + k)$
2D L_∞ Fast halfplane mode	$O(nr^2)^*$	$O(nr)^*$	$O((n/r) \operatorname{polylog} n)$

Table 1: Computational complexities of the implemented algorithms. * indicates a deviation from the reference material.

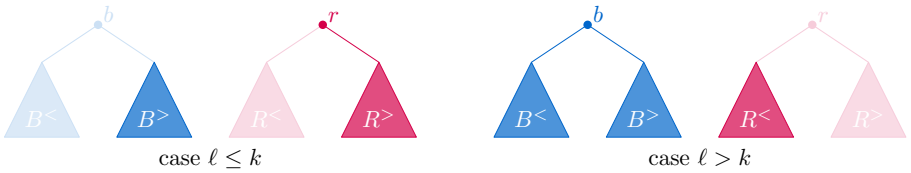


Figure 1: (Sub)Trees that may contain the kth furthest point from q during fast 1D range finding based on $\ell = |B^<| + |R^<| + 1$

2.1 1D

2.1.1 Naive range finding

Preprocessing ($O(n \log n)$ time, $O(n)$ space): Sort the pointset P, save this sorted list.

Query ($O(\log n + k)$ time): Binary search to find the index of the point closest to the query point q within the pointset P. Using this index q_i, then find the kth nearest point by stepping either left or right from q_i in the index range $[q_i - k, q_i + k]$. Return the distance between q and the kth nearest point.

2.1.2 Fast range finding

Based on the algorithm described in section 2.1 of the paper by Van der Horst et al. [HLS22].

Preprocessing ($O(n \log n)$ time, $O(n)$ space): Sort the points, then save points in a node-valued binary search tree T that contains size annotations.

Query ($O(\log n)$ time): Based on the query point q, split T into two trees $T_{P<}$ and $T_{P\geq}$ that contain the points $< q$ and $\geq q$ respectively. Note that both these trees have their points ordered on distance from q, however orderings are flipped between the two trees.
Now, let R be the tree whose root is farther from q, and let B be the other tree. For R, let r be its root, $R^<$ be the subtree containing elements closer to q and $R^>$ be the other tree. Let B have identical definitions for b, $B^<$ and $B^>$. Lastly, let $\ell = |B^<| + |R^<| + 1$.
Based on this ℓ, we now have two distinct cases. These are visualised in Figure 1, which appeared as Figure 3 of van der Horst et al.
$\ell \geq k$: target element is not r or in $R^>$; let $R \leftarrow R^<$
$\ell < k$: target element is not b or in $B^<$; let $B \leftarrow B^>$, and let $k \leftarrow k - (|B^<| + 1)$.
After the modification of the trees has taken place, ensure that r is still farther from q than b; if not, flip the assignments of R and B to ensure this holds. Continue the process until one of the trees is a leaf, after which the kth element can be found trivially using the size annotations.

2.1.3 Naive mode finding

Preprocessing ($O(n \log n)$ time, $O(n)$ space): Sort the pointset P, save this sorted list.

Query ($O(\log n + k)$ time): Based on q and the provided radius, determine the indices into P that fall within the radius around q using a binary search for the lower and upper bound. Afterwards, count the frequency of each color within our index range using a dictionary. Return the color with the highest counted frequency.

2.1.4 Fast mode finding

Based on the algorithm described in section 3 of the paper by Chan et al. [Cha+14].

Preprocessing ($O(n^{3/2})$ time, $O(n)$ space): Transform an array $\bar{A}$ in which each entry contains a color in the range $[0, \Delta)$ into a set of arrays Q_a for $a \in [0, \Delta)$ such that Q_a contains an ordered list of indexes into

$\bar{A}$ where $\bar{A}[i] = a$. Additionally, create an array $\bar{A}'$ such that $\bar{A}'[i]$ is equal to the rank or index of i in $Q_{\bar{A}[i]}$. Lastly, precompute the modes of spans of elements with length $t = \sqrt{n}$. Do this by storing two tables S and S' each of size $t \times t$ such that for any $0 \leq b_i \leq b_j < t$, $S[b_i, b_j]$ contains the mode color of $\bar{A}[b_i t : (b_j + 1)t)$, and $S'[b_i, b_j]$ contains the corresponding frequency. These precomputed spans can be determined using a naive counting implementation.

Query ($O(\sqrt{n})$ time): Given a query range $[i, j)$, calculate $b_i = \lceil i/t \rceil$ and $b_j = \lfloor j/t \rfloor - 1$, representing the indices of the first and last precomputed spans fully within the query range. Let the range $[i : \min b_i t, j)$ (in range elements before the first precomputed span) be known as the prefix, and let the suffix be defined similarly as $[\max (b_j + 1)t, i : j)$.
By Lemma 2 of the paper by Chan et al. the mode of $\bar{A}[i, j)$ must either be an element in the precomputed range defined by b_i and b_j, or an element in the prefix or suffix. Let c be the mode and f_c the corresponding frequency of the precomputed range. Now sequentially scan the elements in the prefix and suffix to see whether these candidates c and f_c need to be updated in order to represent the mode of the whole range $[i, j)$.
Starting from the first element in the prefix, if the color has not yet been considered, determine whether its frequency is at least f_c by testing if $Q_{\bar{A}[x]}[\bar{A}'[x] + f_c - 1] < j$, where x represents the index of the current element. If this is the case, determine the frequency f_x of $\bar{A}[x]$ in $[i, j)$ by doing a linear scan of $Q_{\bar{A}[x]}$ starting from $A'[x] + f_c - 1$. Section 3.3 of their paper shows that this scan is performed in $O(t)$ time. Our candidate c and f_c can then be updated to be $c \leftarrow \bar{A}[x]$ and $f_c \leftarrow f_x$.

2.2 2D - L_∞

2.2.1 Naive range finding

Preprocessing ($O(n)$ space): None, only saving the pointset P.

Query ($O(n \log n)$ time): Determine the distance between all points in P and q. Sort the list, and return the kth entry.

2.2.2 Fast range finding

Based on the algorithm provided in Section 3.1 of the paper by Van der Horst et al. [HLS22].

Preprocessing ($O(n \log n)$ time, $O(n \log n)$ space): Create a rangetree R with size annotations on our pointset P. Additionally, sort P into two arrays A_x and A_y, where A_x contains all points sorted on x coordinates and A_y contains all points sorted on y.

Query ($O(\log^2 n)$ time): Let $x_0, ..., x_\ell$ be the x-coordinates that are at most q_x. Let x_i be one of these coordinates; let $r = q_x - x_i$. We can now find the amount of points in the bounding square around q defined by x_i by doing a counting query using R with the range $[q_x - r, q_x + r] \times [q_y - r, q_y + r]$. Using this count, binary search over $x_0, ..., x_\ell$ in order to find the bounding box with smallest r that contains at least k points. Repeat for the x coordinates greater than q_x as well as the smaller and larger y coordinates, then return the smallest r.

2.2.3 Naive mode finding

Preprocessing ($O(n \log n)$ time, $O(n \log n)$ space): Create a rangetree R on our pointset P.

Query ($O(\log n + k)$ time): Given a radius r, query R for the points in $[q_x - r, q_x + r] \times [q_y - r, q_y + r]$. Determine the mode color of these points by counting using a dictionary and returning the color with highest frequency.

2.2.4 Fast halfplane mode finding

Note: the algorithm provided here does *not* find the mode of a k-nearest neighbour query; it instead returns the mode color of a halfplane query. More details on the algorithm and the reason for this choice are provided in Section 3.2.
Adapted from the algorithm provided in Section 4.2 of the paper by Van der Horst et al. [HLS22].

Preprocessing ($O(nr^2)$ time, $O(nr)$ space): Let L be the set of dual lines corresponding to our initial pointset P. Using a random subset $L_{\mathcal{A}}$ of $r \log r$ lines in L, create an arrangement $\mathcal{A}$ and triangulate it. By Theorem 14 of Chazelle and Friedman [CF88], $\mathcal{A}$ has a constant probability of being a $1/r$ cutting of L. If it is not, retry with different sets $L_{\mathcal{A}}$ until a $1/r$ cutting $\mathcal{A}$ is found.
A conflict list C can then be constructed, in which for each face a in $\mathcal{A}$, C_a stores references to all lines in L that intersect that face. If the maximum amount of lines intersecting a faces is more than n/r, the cutting is rejected and we start over, otherwise we accept the cutting.
Precompute a trapezoidal decomposition $\mathcal{T}$ of $\mathcal{A}$ in order to answer point queries efficiently. Annotate each face a in $\mathcal{A}$ in order to store the mode color m and corresponding frequency f_m of the colors of C_a. Additionally, let L_a be the set of lines that lie fully below a face a. Add the frequency f_c in L_a of any color c that occurs both in L_a as well as C_a to our annotation of a as well.

Query ($O((n/r) \operatorname{polylog} n)$ time): Given a halfspace h in our primal space, transform it to its point dual q_h. Find the face a that contains q_h by performing a point location query on $\mathcal{T}$. Using the annotations on the face a and the conflict list C_a, count the colors of the lines in C_a that are below q_h. Combine these with the precomputed frequencies f_c, and return the mode.

3 Implementation details

In this section, we discuss implementation details for the selected algorithms along with modifications made to the reference material used in order to make implementation easier. All algorithms were implemented in C++11 due to low computational overhead and support for existing libraries containing base geometric algorithms. The implementation itself is freely available on GitHub, along with tools for generating test instances [Pla24].

3.1 1D

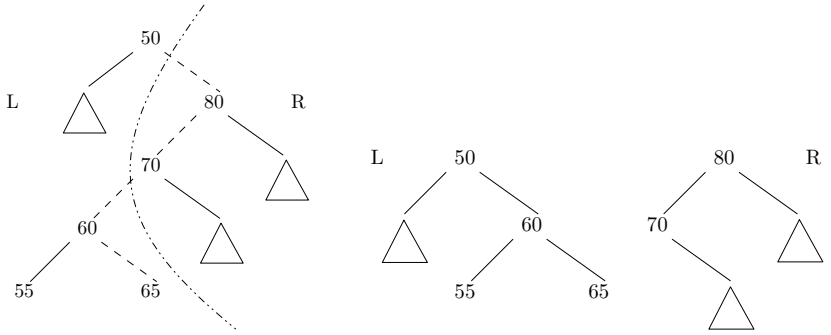
Large parts of the work of Van der Horst et al. [HLS22] were directly implemented, however two elements of note were adapted from the original source material in order to make computation easier and feasible. These were 1) the tree splitting that is required for the range finding operation, and 2) certain details of the fast mode finding algorithm described by Chan et al. [Cha+14].

3.1.1 Tree splitting

Van der Horst et al. requires that the initial ordered tree of points is split into two halves [HLS22]: one containing all points $\geq q$, and one containing all points $< q$. This operation must be performed in $O(\log n)$ in order to maintain the given time complexity of the algorithm. A red-black tree is offered as one possible solution to do this in $O(\log n)$ time, however in practice this is complicated to implement and a computationally heavy operation.

As a sidenote in the original paper, it is mentioned that with some care the same operation could also be performed using a balanced binary search tree. Implementation details for this split operation are however not described. As this still seemed like the better of the two options, we decided to go with this route, and have provided both a methodology for this split as well as a proof of its functionality and time complexity. An illustration of this splitting operation has been provided in Figure 2.

Let T be a node-valued balanced binary search tree containing the points P (where $n = |P|$), and let q



(a) Initial tree with search path and split highlighted

(b) Trees after split

Figure 2: Splitting process on part of balanced binary search tree for $q = 68$. Subtrees unaffected by the split are represented as triangles.

be the query point. Additionally, let $S = [(t_1, d_1), ..., (t_s, d_s)]$ be the search path that is traversed when searching for q in T, where t represents a node that is visited, and d represents the direction in which the search is continued from t (either left or right). As the search stops at t_s, let d_s be the direction in which the search would have continued if t_s had child nodes. Our goal is to create two binary search trees L and R such that L contains all points $< q$, and R contains all points $\geq q$. Furthermore, the height of both L and R must be at most $O(\log n)$.

Using our search path S, we can split T. Let l and r be references to the largest node in L and smallest node in R respectively, and let them both be initialized as a leaf. Furthermore, during our process if l or r are not themselves leaves, let us require that $l.right$ and $r.left$ respectively be a leaf. Now, starting from (t_1, d_1), first determine on which side of the tree the node t_1 will be appended. In general, if a d_i is right, its corresponding t_i must be part of the L and vice versa. Without loss of generality, we will assume that d_i is right in our discussion of how to append t_i to its split tree.

In order to add t_i to L whilst constructing a tree that is still valid binary search tree, we wish to maintain that
1) L only contains points $< q$
2) L is sorted
3) l is the largest node currently present in L
4) $l.right$ is a leaf if l is not a leaf
In order achieve this, we consider two distinct cases:

If l is still a leaf, we can maintain the desired properties by setting $t_i.right \leftarrow leaf$, $l \leftarrow t_i$; all four properties follow trivially from our operation, as $t_i.left$ by definition of a binary search tree could only contain values smaller than t_i and no further reordering is done.
Otherwise, l must by our previous definition already be a node with $l.right = leaf$, and by definition of L we must have gone right in our search path after visiting l. We therefore know that t_i must be larger than l. Additionally, as t_i is part of the same search path as l, all elements in $t_i.left$ must be larger than l while being smaller than t_i. If we therefore set $t_i.right \leftarrow leaf$, $l.right \leftarrow t_i$, followed by updating our reference to l by setting $l \leftarrow t_i$ we know that once again all properties hold. This same process can be applied when d is left by flipping all directions and references to l and L with r and R respectively.

By doing this, we create two trees L and R which maintain their sorted property. We also know that no nodes are lost in this process, as the child that is set to be a leaf during our iterations also is the next node considered for appending to either tree, thereby including it. Furthermore, by definition of a balanced binary search tree, we know that the ith node in the search path in T can have a height of at least $\log n - i$ and at most $\log n - i + 1$. Any additions of nodes are always done on an empty side of a node which came

previously in the search path; the height of the non-empty side was therefore at least $\log n - i$. Using this we can conclude that over a maximum of $\log n$ additions, the overall height of the tree remains $O(\log n)$.
Lastly, in this process only the children of the nodes in the search path are modified. Saving the original nodes in a seperate datastructure such as an array thus allows us to revert the process in $O(\log n)$ time by replacing the nodes in the search path. With this, the following lemma follows:

Lemma 1 *In $O(\log n)$ time, we can split a balanced binary search tree T with n nodes along a query point q into two binary search trees L and R, where L contains all points $< q$ and R contains the points $\geq q$. Furthermore, both L and R have a height at most $O(\log n)$, and the splitting operation can be reverted in $O(\log n)$ time.*

The implementation of the algorithm described by Van der Horst et al. also requires size annotations in each of the tree nodes. In this, we note the following observation:

Observation 1.1 *When splitting T into L and R, the size of a node only changes if that node is part of a search path.*

This follows from the fact that only nodes in the search path have their children modified during the splitting process; As the size of one child does not influence the size of its sibling, we can also conclude that this effect remains limited to the nodes among the search path.

It is therefore possible to both set and revert the new sizes of the nodes in the split trees L and R in $O(\log n)$ time, allowing us to extend our previous Lemma:

Lemma 2 *In $O(\log n)$ time, we can split a balanced binary search tree T with n nodes and size annotations along a query point q into two binary search trees L and R, where L contains all points $< q$ and R contains the points $\geq q$. Furthermore, both L and R have a height at most $O(\log n)$, and the splitting operation can be reverted in $O(\log n)$ time.*

Using this Lemma, we can conclude that it is indeed possible to replace the recommended red-black tree with an in place modified balanced binary search tree.

3.1.2 Fast mode finding

The algorithm described by Chan et al. [Cha+14] was used by Van der Horst et al. in order to achieve a $O(\sqrt{n})$ mode query time with $O(n)$ storage. For this, the first algorithm described in section 3 of Chan et al. suffices. During implementation of this, a few things of note were changed.

Firstly, the array A (and its derivatives) containing all colors were changed from being 1-indexed to being 0-indexed. While this does not have any effect on the functionality or runtime of the algorithm itself, it is noted that some details might differ between the reference description and the eventual implementation as a result. The summary of the algorithm that was provided in Section 2 has already been updated to reflect this change in indexing.

Secondly, in section 3.2 of Chan et al. two indices are calculated: these indices, called b_i and b_j, represent the index of the first and last precomputed mode span respectively in the datastructure S. These are subsequently used in order to determine the mode color of the precomputed part of the query span. We note however that if a query is fully contained within a single precomputed span, these indexes either represent an empty range, or are invalid.
In these cases, a fallback to the naive counting implementation was added instead. We note that this does not change the time complexity of the overall algorithm, as the size of a precomputed span is at most $\sqrt{n}$, therefore any fallback will still run in $O(\sqrt{n})$ time. We additionally note that this implementation also prevents b_j from becoming a negative index in the event that the end of the query span is in the first precomputed span.

3.2 2D

The 2D algorithms described in van der Horst et al. required the use of several well known geometric datastructures and algorithms. As implementing these is prone to subtle issues, requires a significant level of expertise and is generally a lot of work, the decision was made to use an existing library to act as a supporting framework. CGAL 6.0 [Hem24] was selected for this, as it provides most of the datastructures and querying algorithms that we will need to proceed. This allowed us to focus on the novel ideas proposed in the paper.

For range finding, Van der Horst et al. their description for the algorithm could once again be followed. Mode finding on the other hand could not directly be implemented. The reason for this was a limitation in CGAL itself. The original intent was to use the library in order to generate 3D arrangements from planes, as this forms the basis of the precomputed datastructure used for answering queries in the original work. When initially selecting this library however, we failed to notice that only 2D arrangements were supported. The decision was therefore made to implement a alternate version of the mode query which works on query halfplanes instead.

The alternate version uses much of the same machinary seen in the original, however it has been adjusted to work with 2D arrangements instead. By providing this alternate version, we hope to both get a general idea of the performance of mode queries using this basis, as well as providing a framework which can be expanded upon when 3D arrangements can be easily made.

A last thing of note is that the introduction of a second dimension can lead to degeneracies such as duplicate x or y coordinates. Due to time constraints, no expansive validation was performed to ensure that our implementation could handle all edge cases caused by this. As a workaround, datapoints are slightly offset in order to discourage degeneracies from forming.

3.2.1 Fast range finding

As noted before, the range finding algorithm was directly implemented as described by Van der Horst et al. The method requires the use of a range tree for counting queries. Range tree implementations are widely available, therefore one was selected that matches our algorithmic requirements. As we aim to fully remove dependency on k and aim to match the runtime described in the original paper, a range tree that supports fractional cascading and native counting queries was required. CGAL, the library used for most of the base geometric algorithms in 2D mode finding, has a d-dimensional range tree with fractional cascading [Ney24]. It however sadly does not support native counting queries, therefore the use of this would introduce dependency on k as iterating over output points is required.

An alternative was found in an open source project by Weihs, which implemented d-dimensional range trees based on doubles [Wei20]. As we wanted to integrate these results with the CGAL implementation used in mode finding, the range tree implemented by Weihs was modified in order to support native CGAL points [Hem+24] [Hem24] [See24].

3.2.2 Fast mode finding

A 2D arrangement version of the algorithm described by Van der Horst et al. was created. In the original work, points are passed through a lifting operation in order to create planes. A 3D arrangement using $r \log r$ of these planes can then be created and triangulated, thereby creating a $1/r$ cutting of the dual space with constant probability. They then show that a query point q and radius r can be transformed into a point q^* in the dual space, where the mode of points within r distance from q can be found as the mode of the planes below q^* in the dual space.

In order to achieve a similar result using only 2D arrangements, the lifting operation was replaced with a simple point/line dual. This once again allows us to create a $1/r$ cutting in dual space, where the lines below a point q^* can be found using the same mechanisms as in 3D. The query point q^* now maps to a halfspace in the primal space, therefore making this alternate algorithm a halfspace mode query.

In order to generate face conflict lists and color frequencies per face as mentioned in Section 2.2.4, a naive method was used which directly compares all faces with all lines. The resulting preprocessing time is therefore $O(nr^2)$ as can be seen in Table 1, which is greater than the theoretical runtime of $O(n^{1+\delta} + nr^3)$ described by van der Horst et al. Due to storing the entire conflict list and color frequencies instead of computing them at runtime, our space requirement is also $O(nr)$ instead of the theoretical $O(n + r^3)$. Both of these could still be improved with a better implementation.

CGAL was used for line and segment intersections, point representation, 2D arrangement creation and annotation and trapezoidal decompositions of the arrangement for point location queries [Hem24] [Hem+24] [Wei+24]. The rest of the required datastructures and operations were implemented using features natively available to C++11.

4 Computational results

In this section, we aim to provide an overview of the test instances used to validate performance of the implemented algorithms. Additionally, performance graphs are shown for artificial as well as real life based instances. For a full overview of unformatted results, we refer the reader to the Appendices.

All results were measured using C++11 in Visual Studio on Windows in release mode using default compiler settings, and run on a Intel i7-10750H CPU using 40GB of RAM.

4.1 1D

For 1D, both artificial and real life data are used in order to perform computational experiments. The use of artificial data for this section is largely due to the fact that real life data often has more than a single dimension, making the acquisition of a comprehensive dataset difficult. Some real life data is included in order to validate the results that were achieved in the artificial counterpart, however this is based on an originally 2D dataset which is projected in order to match our requirements.

Artificial data

Both the naive and tree-based range finding approaches are not dependent on the spatial distribution of sample points, as both methods work in rank space. Therefore, a simple uniform distribution in the arbitrarily chosen range [-50000, 50000] is used in order to generate the location of both the sample as well as the query points.

The color of the points does have influence on the runtime of the mode determination; The naive case is unaffected, however the faster implementation mentioned in Chan et al. requires less steps when a color appears multiple times in the prefix and suffix of the mode interval [Cha+14]. Therefore, two different generation methods are employed:

- Uniformly sampled colors in the integer range [0, Δ). This represents a situation in which the position of the point has no correlation its color.
- $\gamma \leq n$ random points are selected and given a color uniformly sampled from [0, Δ). Then, for all other points, their color is the same as the closest point with a chance α, or randomly sampled from [0, Δ) otherwise. This models a situation in which clusters of colors are present, however a certain degree of variability is still included. Note that the previous case can also be modeled this way by setting $\gamma = \alpha = 0$.

Using these methods, the test scenarios seen in Table 2 were generated. In this, we use a scenario naming scheme of 1D-$\{n/1000\}$-k-Δ-γ-$\{\alpha \cdot 100\}$.

Scenario	n	Δ	γ	α
1D-1-k-20-0-0	1,000	20	0	0
1D-10-k-20-0-0	10,000	20	0	0
1D-100-k-20-0-0	100,000	20	0	0
1D-1-k-100-0-0	1,000	100	0	0
1D-10-k-100-0-0	10,000	100	0	0
1D-100-k-100-0-0	100,000	100	0	0

(a) Uniform color scenarios

Scenario	n	Δ	γ	α
1D-1-k-20-30-95	1,000	20	30	0.95
1D-10-k-20-30-95	10,000	20	30	0.95
1D-100-k-20-30-95	100,000	20	30	0.95
1D-1-k-100-150-95	1,000	100	150	0.95
1D-10-k-100-150-95	10,000	100	150	0.95
1D-100-k-100-150-95	100,000	100	150	0.95

(b) Clustered color scenarios

Table 2: Overview of 2D artificial scenarios

For each of these scenarios 10 unique data sets were generated. On these, testing is done using $k = \{10, 25, 50, 75, 100, 520, 500, 750, 1000, 1500, 2000\}$ (with $k \geq 1000$ only on scenarios with $n > 1000$) and $Q = 1000$ uniformly sampled query points. Results across the data sets of each scenario are averaged in order to produce computation times in milliseconds for each of the operations.

In addition to these scenarios, a seperate test was run in which the effects of k as a fraction of n were determined. This was done by doing an performing additional computations on 1D-10-k-100-0-0 and 1D-10-k-100-150-95, in which k was taken to be $0.1n, 0.2n, ..., 0.9n$. The number of runs was also reduced from 10 to 5, in order to reduce the computational load for this set of tests. We note that this might increase the variability of the results, however as we are only looking for a rough trend this was deemed acceptable.
Other than that, parameters for the scenarios were kept the same. 1D-10-k-100-0-0 and 1D-10-k-100-150-95 were arbitrarily selected for this process, however we note that they both have a reasonable instance size ($n = 10,000$) and did not show any major deviation from the performance of the other scenarios. We therefore felt that these would be suitable in order to run this additional experiment.
To maintain a clear separation within the results overview, these tests are labeled 1D-10-k-100-0-0-FRAC and 1D-10-k-100-150-95-FRAC for the runs concerning 1D-10-k-100-0-0 and 1D-10-k-100-150-95 respectively.

Real life data

For the real life points, low dimensional data was gathered and projected to 1D. The primary source used is community gathered temperature data provided by the UK MetOffice's WOW project [WOW24], which is distributed under the open government licence.

Geolocated temperature data for a total of 10 days was used, spanning from 02-06-2024 to 12-06-2024, where each day consisted of a approximately 6200 data points. For each of these points, a color was determined by binning temperature values in 2.5°C intervals, resulting in a total of 21 possible colors. This resulted in a map such as the one displayed in Figure 3. Note that for the sake of visual clarity, this view of the collected data is cropped to only include Europe. Large amounts of sample points are also present in the continental United States and Oceania, and are sparse elsewhere.

Data was then projected along the longitude (y-axis) or latitude (x-axis) in order to get a 1D dataset; We shall call these datasets 1D-TMP-LON-k and 1D-TMP-LAT-k respectively. Afterwards, testing was once again done using $k = \{10, 25, 50, 75, 100, 520, 500, 750, 1000, 1500, 2000\}$ and $Q = 1000$ uniformly sampled query points, and results were averaged over the 10 days.

Results

For each scenario, the average time in milliseconds over the 10 runs is given for 5 distinct operations:

- The building of the tree datastructure for range queries. We note that at time of writing, this could still be significantly improved by better memory allocation procedures.
- Executing $Q(= 1000)$ random range queries using the tree approach.

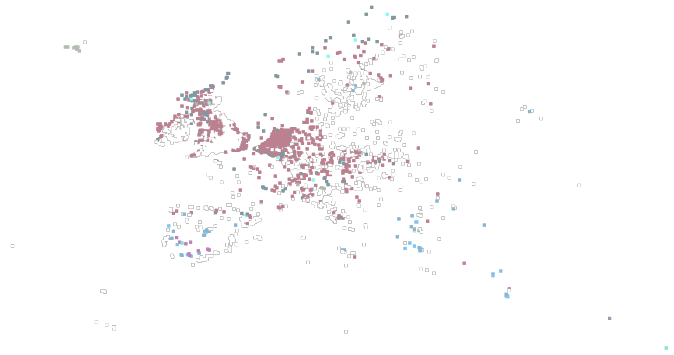
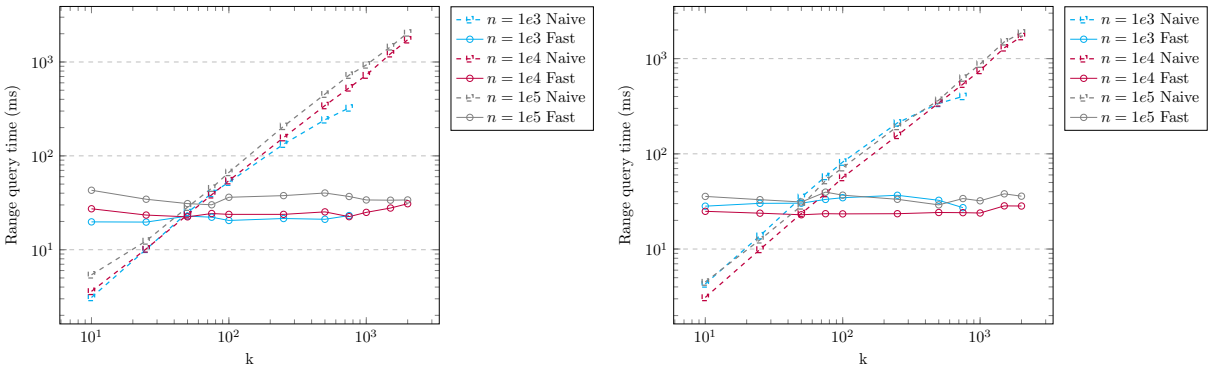


Figure 3: Temperature data from 2024-06-02, cropped to Europe

- Executing Q random range queries using the naive approach.
- Executing the corresponding Q mode queries using the fast approach.
- Executing the corresponding Q mode queries using the naive approach.

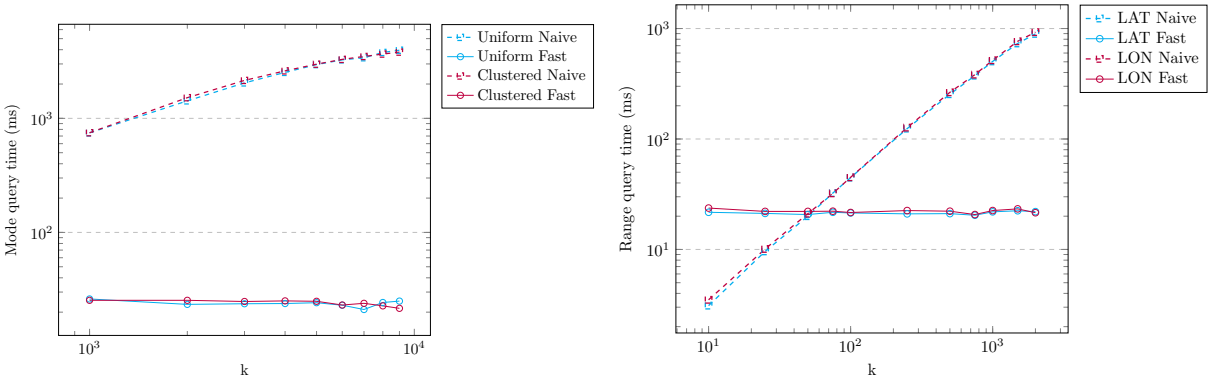
The results for range and mode queries have been compiled into graphs. Additionally, the raw tables can be found in the Appendix. As results between groups of instances did not show large differences, a representative selection is displayed instead of all individual instances.



(a) 1D-{1,10,100}-k-20-0-0

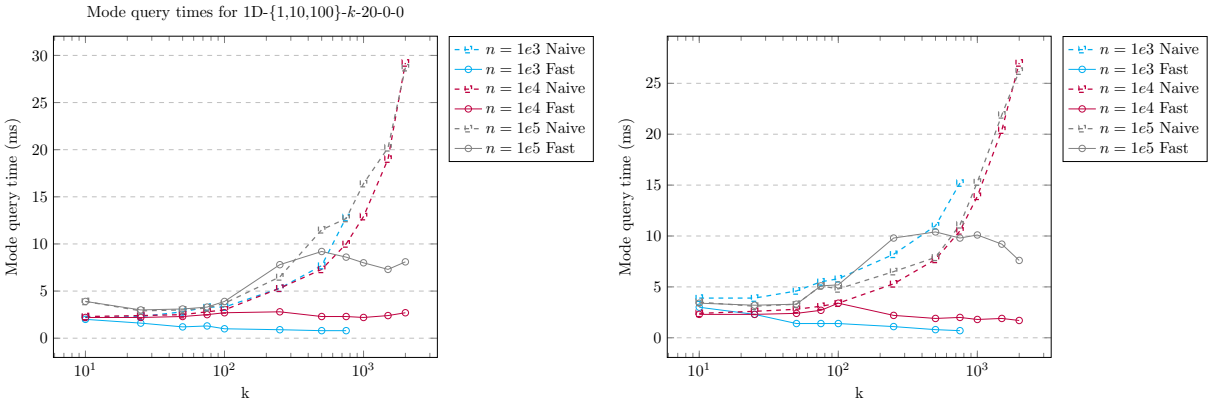
(b) 1D-{1, 10, 100}-k-100-150-95

Figure 4: Naive and fast range query times on 1D artificial instances



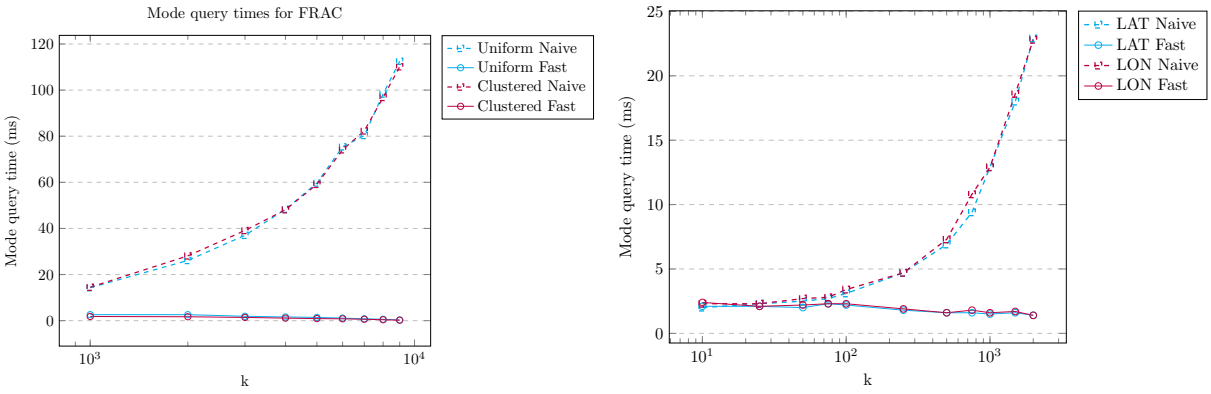
(a) FRAC uniform and clustered (b) 1D-TMP-{LAT,LON}

Figure 5: Naive and fast range query times on 1D FRAC and TMP instances



(a) 1D-{1,10,100}-k-20-0-0 (b) 1D-{1, 10, 100}-k-100-150-95

Figure 6: Naive and fast mode query times on 1D artificial instances



(a) FRAC uniform and clustered (b) 1D-TMP-{LAT,LON}

Figure 7: Naive and fast mode query times on 1D FRAC and TMP instances

Discusssion

For range queries, the results we see are largely expected. In Figure 4 we see that the naive implementation follows a roughly linear relation between k and the query time, matching our expectation of $O(\log n + k)$. In Figure 5a, we do see that computation time of the naive approach seems to taper off for larger k values. This is most likely due to the fact that a large part of the window size for sorting is outside of array bounds as $2k > n$, thus reducing the amount of computation required. When looking at the range queries as described by Van der Horst et al., expectations are again met. Query time stays roughly the same over all values of k, and on average we see a slight increase in query time for scenarios with a larger n. We note that it appears that the overhead associated with the tree query is rather high, as can be seen due to the minor query time differences between $n = 1000$ and $n = 100,000$. In general, we see that when $k \gtrapprox 100$ the fast method seems to outperform the naive in all given figures.

For mode, results are also mostly in line with expectation. We do note that in Figure 6 we seem to see an initial jump in computation time of our fast method for large k with $n = 1e5$, after which computation time seems to stabilize or even slightly decrease for even larger k values. When investigating this, this bump in the curve is most likely due to a high initial overhead for use of the precomputed structure (when compared to doing queries that fall completely within a single precomputed span, thus using a naive counting approach). As k becomes larger, this initial overhead is less pronounced causing a flattening of the curve. In general for mode, when $k \gtrapprox 300 - 500$ our fast approach seems to be better than the naive.

4.2 2D

A mix of artificial and real life data was once again used in order to validate results. As 2D datasets are generally more widely available, a larger selection is included here as opposed to the 1D instances.

As preprocessing times for the datastructures required for the 2D algorithms were relatively high, graphs for these were also included.

Artificial data

The same method of generating data that was used in 1D was generalized for 2D. We refer back to the 1D section for more details, however a couple of small changes are highlighted.

- Points were once again sampled in an arbitrarily selected range. Seeing as we are now in 2D, this range has been expanded to $[-50000, 50000] \times [-50000, 50000]$.
- For the clustered color scenarios, the distance measure used for determining which color a point gets is L_∞. This was chosen due to the fact that this is also the metric used in the algorithms themselves.

For our mode operation, we now also have an additional parameter r. In van der Horst et al. this value was chosen to be $\sqrt{n}$ in order to get a $O(\sqrt{n}\,\text{polylog}\,n)$ query time, however during testing we found that preprocessing using our current implemention was exceedingly slow. We therefore decided to use a small constant r instead for all artificial instances. In order to determine a suitable value, the scenario 2D-10-25-r-20-0-0 was selected and run with $r = \{5, 10, 25, 50\}$. Larger values than this were not tested, as preprocessing times would exceed 1 hour per run of an instance. A balance between preprocessing and query time was then selected by using $r = 10$, which required 6 seconds of preprocessing time for a single run. We refer to the raw results in the Appendix for more details on this experiment. Experiments including variable r will be discussed in more detail for the real life instances.

An overview of the tested artificial scenarios can be seen in Table 3. In this, we use a scenario naming scheme of 2D-$\{n/1000\}$-k-r-Δ-γ-$\{\alpha \cdot 100\}$.

Scenario	n	Δ	γ	α
2D-1-k-r-20-0-0	1,000	20	0	0
2D-10-k-r-20-0-0	10,000	20	0	0
2D-100-k-r-20-0-0	100,000	20	0	0
2D-1-k-r-100-0-0	1,000	100	0	0
2D-10-k-r-100-0-0	10,000	100	0	0
2D-100-k-r-100-0-0	100,000	100	0	0

(a) Uniform color scenarios

Scenario	n	Δ	γ	α
2D-1-k-r-20-30-95	1,000	20	30	0.95
2D-10-k-r-20-30-95	10,000	20	30	0.95
2D-100-k-r-20-30-95	100,000	20	30	0.95
2D-1-k-r-100-200-95	1,000	100	200	0.95
2D-10-k-r-100-200-95	10,000	100	200	0.95
2D-100-k-r-100-200-95	100,000	100	200	0.95

(b) Clustered color scenarios

Table 3: Overview of 2D artificial scenarios

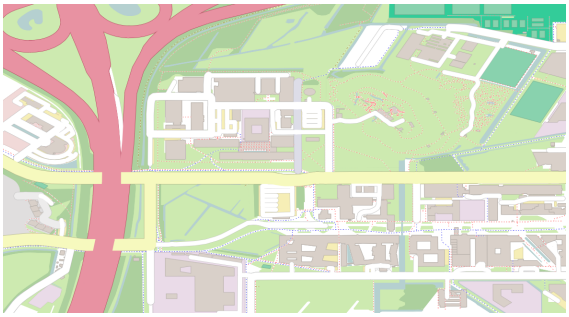
Real life data

As we already have a usable dataset in the form of the weather based one generated for 1D testing, this is reused when testing the 2D implementation. We will refer to this instance as 2D-TMP-k-r. This dataset is used in order to visualise the relation between processing and query time and as a function of r. For these tests, $r = \{2, 5, 10, 15, 20\}$ is used.

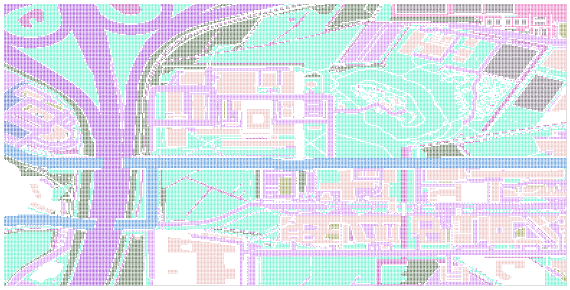
Additional data is however also considered, as 2D datasets are widely available. We use annotated map data as our secondary source of test data. OpenStreetMap [Ope24] was used in order to download map data from 10 places in the Utrecht region, and QGIS [QGI24] was used in order to render a simplified version of the map. See Figure 8a for an example. In selecting test data, attempts were made in order to include regions that showcased a large amount of different colors. A single test instance was also included which was mostly farmland, allowing for validation when the variation in colors is low. For more details on the exact test cases, we refer to the renderings of the selected maps which are available on GitHub [Pla24].
The map renderings were then sampled in order to produce colored points. The location of the point was taken to be the pixel coordinate in the map view image, along with a small randomized offset. This offset was uniformly sampled within a distance of 0.05 from the pixel coordinate, and was included in order to discourage exact x and y coordinate matches for different points. Sampling rate was set such that the total instance size was around $50,000$ points for each of the included maps. An example of the resulting sampled map can be found in Figure 8b.

As the amount of sample points is relatively high, we chose to generate instances based on random subsets of the full point sets. These were then used in order to visualize the dependency on n seen in the algorithms by van der Horst et al. In this, r was once again fixed at 10 as was the case with the artificial test cases. We will refer to the instances as 2D-MAP-k-p, where $0 < p \leq 100$ represents the percentage of total map points used in the instance. $p = \{1, 5, 10, 25, 50, 75, 100\}$ were selected in order to make instances, and each was run with the the same selection of k values as used in the artificial data.

For each instance, all maps were once again run with $Q = 1000$. Results from the different maps were then averaged in order to produce computation times for a certain instance.



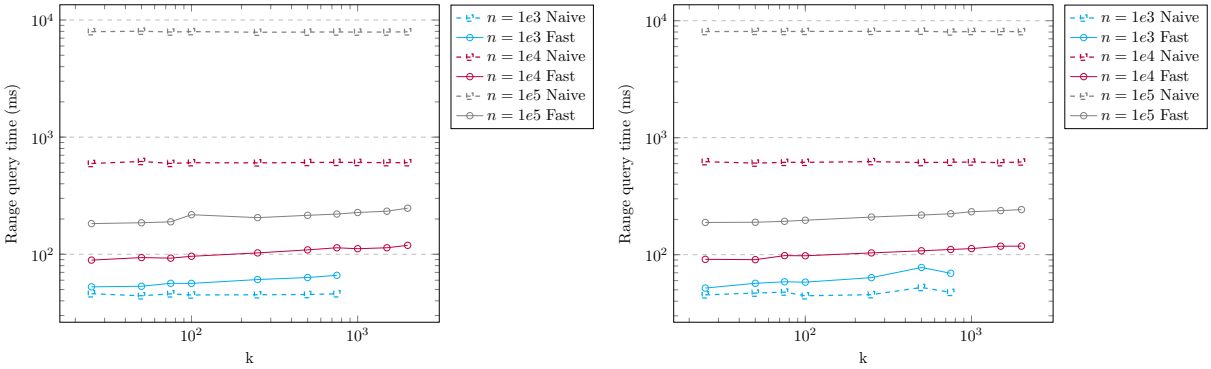
(a) Simplified map view of Utrecht Science Park, centered on the Buys Ballot Building.



(b) Point sampling taking from the simplified map view.

Figure 8: 2D map data to pointset conversion

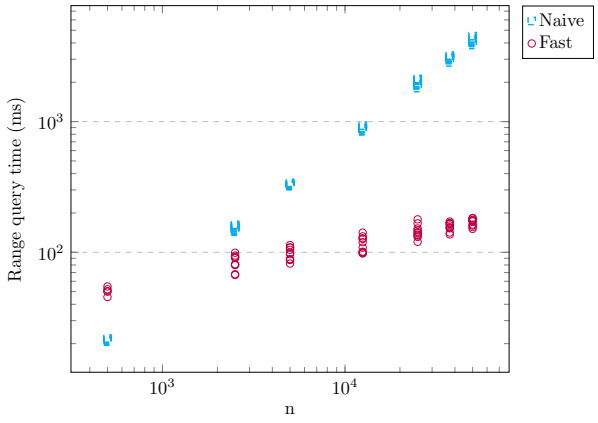
Results



(a) 2D-{1,10,100}-k-10-20-0-0

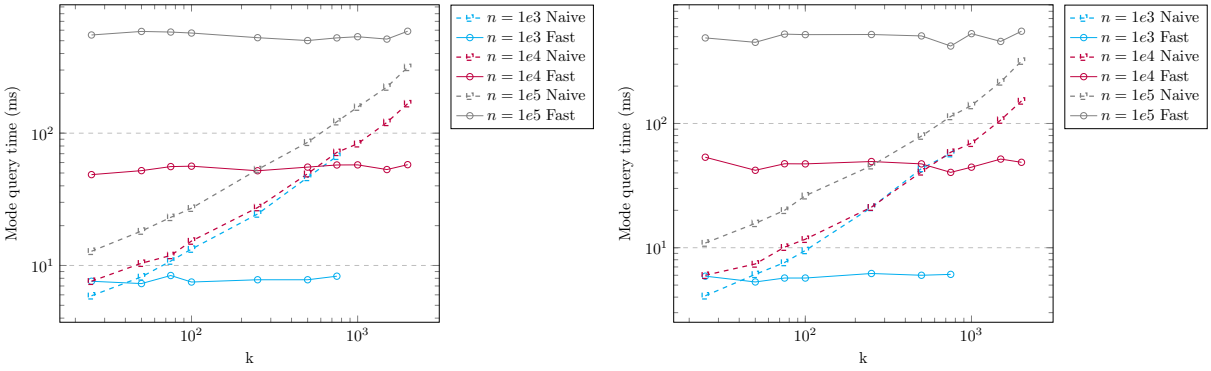
(b) 2D-{1,10,100}-k-10-100-200-95

Figure 9: Naive and fast mode query times on 2D artificial instances



(a) 2D-MAP-k-p, multiple k values displayed for a single n

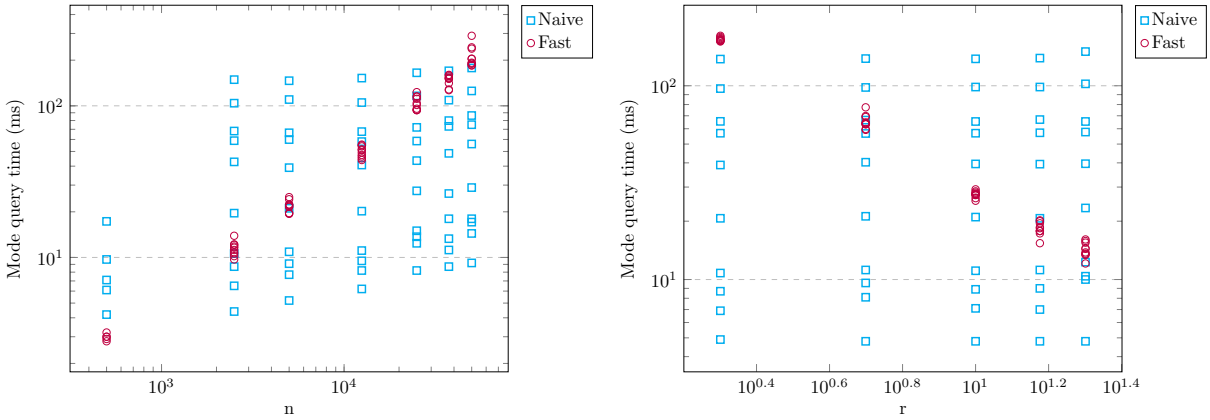
Figure 10: Naive and fast range query times on 2D real life map instances, plotted against n. Temperature instances not considered as range has no dependency on r.



(a) 2D-{1,10,100}-k-10-20-0-0

(b) 2D-{1,10,100}-k-10-100-200-95

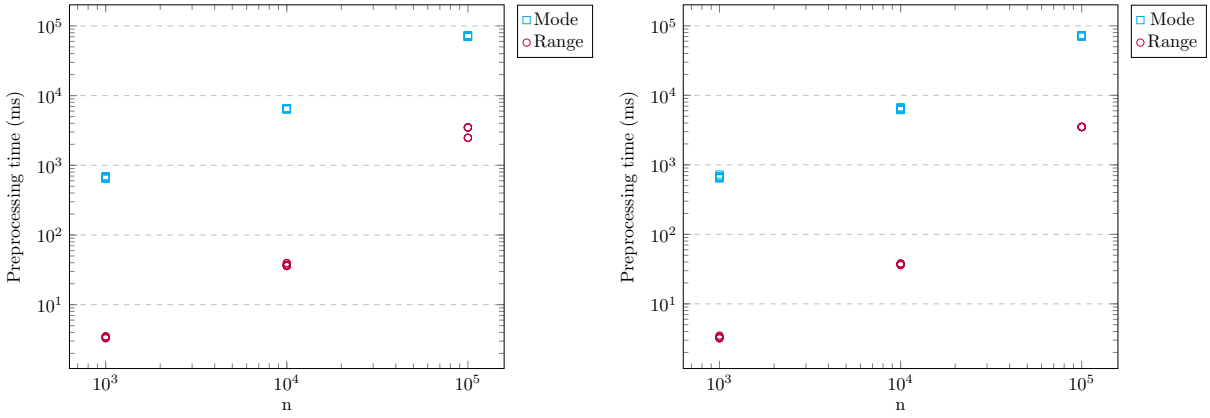
Figure 11: Naive and fast mode query times on 2D artificial instances



(a) 2D-MAP-k-p, multiple k values displayed for a single n

(b) 2D-TMP-k-r

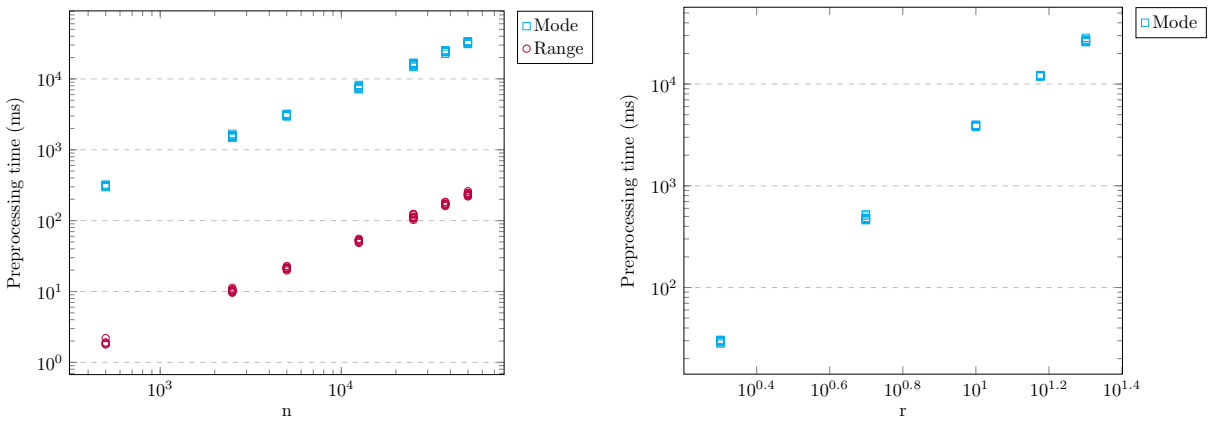
Figure 12: Naive and fast mode query times on 2D real life instances. Plotted against n and r respectively.



(a) 2D-{1,10,100}-k-10-20-0-0

(b) 2D-{1,10,100}-k-10-100-200-95

Figure 13: Preprocessing times of range and mode datastructures on 2D artificial instances. Range and mode are combined into a single graph in order to save space.



(a) 2D-MAP-k-p

(b) 2D-TMP-k-r, $n \approx 6200$

Figure 14: Preprocessing times of range and mode datastructures on 2D real life instances. Range excluded in temperature instances due to no dependency on r.

Discussion

The range query results only somewhat match our expectations. When only looking at Figure 9, the naive implementation seems to only be dependent on n, which could match the theoretical $O(n \log n)$ query time. Looking at Figure 10a, we see a near linear relationship, which all but confirms the theoritical result. Of note is that for $n = 1000$ in Figure 9, the naive approach actually outperforms the fast implementation; we can therefore conclude that the hidden constant for our fast $O(\log n)$ approach must be rather large.
The fast implementation does seem to have a slightly increased query time for larger k; this is best visible in Figure 9b and is also the cause of the vertical spread in Figure 10a. This anomaly was however explained when directly looking at the performance metrics in Visual Studio. We found here that for larger k, more time is spent querying large windows in our range tree. This in turn resulted in longer path traversals within the range tree, causing an increased runtime. As this seems to be the only source of computation time increase between small and large k tests, we can therefore still conclude that our worst case is bounded by $O(\log^2 n)$.

When looking at mode queries, the results directly seem to match our expectations. In Figure 11, we see that the naive implementation shows a linear relation with k with different starting points for our smallest k, matching our $O(\log n + k)$ described algorithm. The query times of the fast implementation seem to be completely independent of k, thereby possibly matching our $O((n/r)\operatorname{polylog} n)$ query time. Looking at Figure 11a we clearly see some dependency on n, and 11b shows the inverse relationship with r. Due to the high dependence on n it is hard to pinpoint an exact k value at wich query times are lower than the naive approach, however it seems to be in the neighbourhood of $0.01n$ to $0.1n$.

Looking at only Figure 13, preprocessing times for both the range and mode datastructures seem to follow a roughly linear relation with n. For the range datastructure this is no surprise, as the small number of sampled n and scale of the graph would make it hard to see the $\log n$ factor that is included in the theoretical $O(n \log n)$ runtime.
As for the mode preprocessing, this following a roughly linear relation is also expected. As r was fixed for all these test cases currently considered, the linear relation matches the expected $O(nr^2)$ time. Only when we look at Figure 14b do we see the additional dependence on r. We once again note that this preprocessing time can most likely be reduced with a better implementation; as it stands, preprocessing for large r is often not worth it due to the amount of queries that would be required to offset the initial workload.

5 Conclusion

In this paper, we have succesfully implemented a selection of algorithms from van der Horst et al. [HLS22]. In doing this, we showed that Chromatic k-Nearest Neighbours can be answered in a computationally efficient manner with no other dependence than the number of input points n, and have provided a base implementation that may be generalized to higher dimensions and other distance metrics.

In our computational analysis, we see clear benefits from the approach proposed by van der Horst et al. In 1D, we see clear improvements in range query times when $k \gtrapprox 100$. For mode queries, we see similar behavior from $k \gtrapprox 300$, thereby allowing us to conclude that the methods implemented might greatly increase performance when compared to naive approaches when k is large.
For 2D we achieve similar results, with a breakeven point for range queries at around $n = 2000$ (regardless of k). For mode queries, we implemented a modified version of the algorithm which returns halfplane queries instead; this also outperforms the naive approach when k is around $0.01n$ to $0.1n$, once again making it useful when k is large.

Some things of note can still be directly improved in our implementation. For 2D mode finding specifically, we deviated both from the theoretically optimal running time as well as space usage by using a naive approach for handling conflict lists. Implementing the improvements described in the original paper here could drastically decrease the preprocessing time of the algorithm, thereby allowing larger r values to be used for even lower query times. If feasible, this would also allow for the theoretical $r = \sqrt{n}$ to be used. Secondly, the implementation as a whole can probably still be improved. More agressive compiler optimizations can still be used given some more testing, and due to a lack of experience with C++11 and CGAL some additional possibilities for speedup were likely overlooked.

Due to time constraints, something that was left out of testing was handling of all edge cases within the implementation. Especially for the 2D implementation, testing should still be done using duplicate data points, overlapping query and data points and grid-like datasets in which x and y coordinates are strictly not unique to validate that the correct result is still returned. In addition to this, experiments may be run with different numerical data types in CGAL in order to quantify the performance decrease associated with higher precision point locations and line intersections.

Finally, once 3D arrangements are available further experimentation should also be done in order to fully implement 2D mode finding. This can then be used in order to generalize to higher dimensions, as well as

solving inherent limitations present in the halfspace queries such as a small central clusters of points never being a mode candidate.

References

[CF88] B. Chazelle and J. Friedman. "A deterministic view of random sampling and its use in geometry". In: *[Proceedings 1988] 29th Annual Symposium on Foundations of Computer Science*. 1988, pp. 539–549. DOI: 10.1109/SFCS.1988.21970.

[Mou+00] David M. Mount et al. "Chromatic nearest neighbor searching: A query sensitive approach". In: *Computational Geometry* 17.3 (2000), pp. 97–119. ISSN: 0925-7721. DOI: https://doi.org/10.1016/S0925-7721(00)00021-3. URL: https://www.sciencedirect.com/science/article/pii/S0925772100000213.

[Cha+14] Timothy M. Chan et al. "Linear-Space Data Structures for Range Mode Query in Arrays". In: *Theory of Computing Systems* 55.4 (Nov. 2014), pp. 719–741. ISSN: 1433-0490. DOI: 10.1007/s00224-013-9455-2. URL: https://doi.org/10.1007/s00224-013-9455-2.

[Wei20] L. Weihs. *Range Tree*. https://github.com/Lucaweihs/range-tree. 2020.

[HLS22] Thijs van der Horst, Maarten Löffler, and Frank Staals. "Chromatic k-Nearest Neighbor Queries". In: *30th Annual European Symposium on Algorithms (ESA 2022)*. Ed. by Shiri Chechik et al. Vol. 244. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 67:1–67:14. ISBN: 978-3-95977-247-1. DOI: 10.4230/LIPIcs.ESA.2022.67. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2022.67.

[Hem24] Michael Hemmer. "Algebraic Foundations". In: *CGAL User and Reference Manual*. 6.0.1. CGAL Editorial Board, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html#PkgAlgebraicFoundations.

[Hem+24] Michael Hemmer et al. "Number Types". In: *CGAL User and Reference Manual*. 6.0.1. CGAL Editorial Board, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html#PkgNumberTypes.

[Ney24] Gabriele Neyer. "dD Range and Segment Trees". In: *CGAL User and Reference Manual*. 6.0.1. CGAL Editorial Board, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html#PkgSearchStructures.

[Ope24] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. https://www.openstreetmap.org. 2024.

[Pla24] T.D. van der Plas. *Chromatic k Nearest Neighbours*. https://github.com/tvdplas/chromatic-k-nearest-neighbours/. 2024.

[QGI24] QGIS Development Team. *QGIS Geographic Information System*. QGIS Association. 2024. URL: https://www.qgis.org.

[See24] Michael Seel. "dD Geometry Kernel". In: *CGAL User and Reference Manual*. 6.0.1. CGAL Editorial Board, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html#PkgKernelD.

[Wei+24] Ron Wein et al. "2D Arrangements". In: *CGAL User and Reference Manual*. 6.0.1. CGAL Editorial Board, 2024. URL: https://doc.cgal.org/6.0.1/Manual/packages.html#PkgArrangementOnSurface2.

[WOW24] Met Office WOW. *Met Office WOW - Home Page — wow.metoffice.gov.uk*. https://www.wow.metoffice.gov.uk/. [Accessed 12-06-2024]. 2024.