A Computational Analysis of a Novel Chromatic k-Nearest Neighbours Algorithm

Thomas van der Plas, Frank Staals, Erwin Glazenburg

10 November 2024

Abstract

TODO: daadwerkelijke cijfers, betere text; dit is nog niet heel denderend Chromatic k-Nearest Neighbour Classification is used to determine the color of a query point based on the mode color of its k closest neighbours. Classical approaches to solving this problem have a runtime which is dependent on k, causing large neighbourhood searches to become expensive. We investigate the computational performance of an alternative solution method in which runtime is independent of k. 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 $k \gtrsim$ ... in 1D and $k \gtrsim$ ... in 2D, validated against both artificial and real life data points.

1 Introduction

TODO: Ik was eerlijk gezegd vergeten dat ik deze nog moest doen, dus deze zal in de uiteindelijke versie komen. Main talking points: What is Chromatic k-Nearest Neighbour and what is it used for, what are known fast methods (based on table in Thijs' report), what will we be implementing, why is that important.

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		
Naive range	$O(n \log n)$	$O(n)$	$O(\log n + k)$
Fast range	$O(n \log n)$	$O(n)$	$O(\log n)$
Naive mode	-	-	$O(k)$
Fast mode	$O(n^{3/2})$	$O(n)$	$O(\sqrt{n})$
	2D - L_∞		
Naive range	$O(n \log n)$	$O(n)$	$O(\log n + k)$
Fast range	$O(n \log n)$	$O(n \log n)$	$O(\log^2 n)$
Naive mode	-	-	$O(k)$
Fast mode	$O(n^2)^*$	$O(nr)^*$	$O(n/r polylog\, n)$

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

2.1 1D

2.1.1 Naive range finding

Precomputation: Sort the pointset P, save this sorted list.

Query: 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 $[q_i - k, q_i + k]$.

2.1.2 Fast range finding

Based on the algorithm described in section 2.1 of the paper by Van der Horst et al. [Cha+14].

Precomputation: Sort the points, then save points in a node-valued binary search tree T that contains size annotations.

Query: 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 three distinct cases:
$\ell \geq k$: 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

Precomputation: None.

Query: Count each color in the provided index range. Return the color with the highest frequency.

2.1.4 Fast mode finding

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

Precomputation: 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: 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 $[\max (b_j + 1)t, i : j)$.
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 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 $Q_{\bar{A}[x]}$. 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

Precomputation 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 A generalized version of the stepping performed in the 1D case. Binary search in x and y in order to find the index of point q in A_x and A_y. From this index q_i, expand the bounding square around q_i k times by stepping towards the closest point in any cardinal direction, taking care not to double count points that might have been already seen when stepping in a different dimension. Return the radius of the bounding square once done.

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].

Precomputation Create a rangetree R 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 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

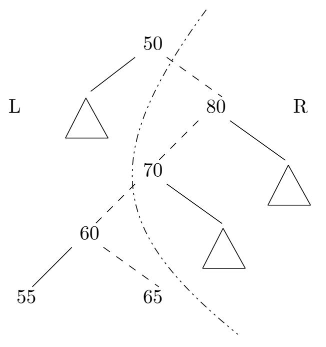
Precomputation None

Query Let $[x_i, x_j]$ and $[y_i, y_j]$ be index ranges into arrays representing the x and y coordinates of the points respectively. For both of these arrays, count the frequency of each color while taking care to not double count points. Return the mode.

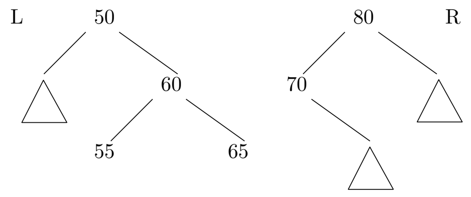
2.2.4 Fast mode finding

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].

Precomputation 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, a $1/r$ cutting can be generated with constant probability. A candidate cutting is generated



(a) Initial tree with search path and split highlighted



(b) Trees after split

Figure 1: Splitting process on part of balanced binary search tree for $q = 68$. Subtrees unaffected by the split are represented as triangles.

by creating an arrangement $\mathcal{A}$ on $L_{\mathcal{A}}$ and triangulating it. A conflict list C can then be constructed, in which for each face f in $\mathcal{A}$, C_f 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.
Using $\mathcal{A}$, precompute a datastructure $\mathcal{T}$ for trapezoidal point queries. Additionally, let L_f be the set of lines that lie fully below a face f. Using this, annotate each face f of $\mathcal{A}$ in order to contain the mode color m and frequency f_m of the colors of C_f, as well as the frequencies f_c of any color c in L_f that also occurs in C_f.

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

3 Implementation details

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 1.

Let T be a node-valued balanced binary search tree containing the points P (where $n = |P|$), and let q 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 out process if l or r are not leaves, let $l.right$ 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 $t.right \leftarrow leaf$, $l.right \leftarrow t$, followed by updating our reference to l by setting $l \leftarrow t$ 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. their publication 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.'s publication, 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

TODO: ik ben niet helemaal blij met waar ik het alternatieve algoritme nu uitleg, maar ik weet ook niet zo goed waar ik het anders kan neerzetten (behalve die algorithm overview in sectie 2, maar dan is een beetje in strijd met het feit dat dat allemaal kort is).

The 2D algorithms described in Van der Horst et al. their work 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 [Wei+24]. 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.

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 is then created and triangulated, creating a $1/r$ cutting of the dual space. 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 precomputation time is therefore $O(n^2)$ as can be seen in Table 1, which is greater than the theoretical runtime of $O(n^{1+\delta}+nr^3)$. 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.

Alternate implementation

4 Computational results

4.1 1D

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

- Uniformly sampled colors in the integer range $[0, \Delta)$. 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 $[0, \Delta)$. Then, for all other points, their color is the same as the closest point with a chance α, or randomly sampled from $[0, \Delta)$ 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 following test scenarios were generated.

Uniform color scenarios				
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

Clustered 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

For each of these scenarios 10 unique data sets were generated. On these, testing was 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 were 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.
In order to maintain a clear separation within the results overview, these tests

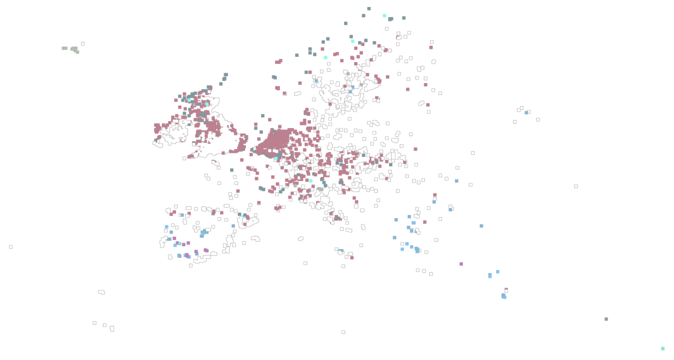


Figure 2: Temperature data from 2024-06-02, cropped to Europe

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 5°C intervals. This resulted in a map such as the one displayed in Figure 2. 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.

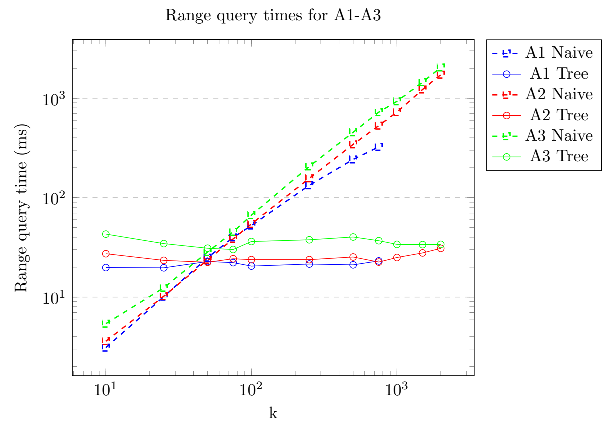
Results

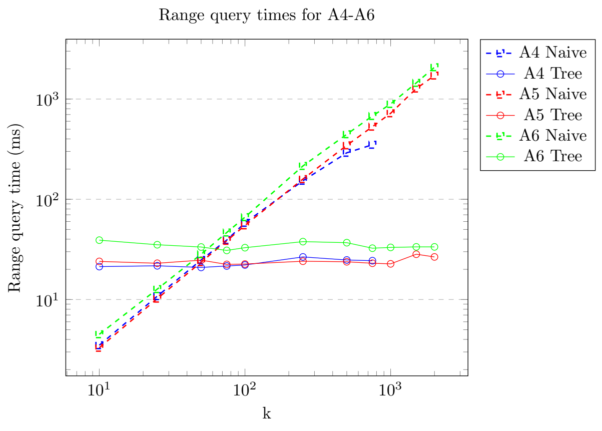
Computation time was measured for all scenarios mentioned above. Implementation was done in C++11, and run on a Intel i7-10750H using 40GB of RAM.

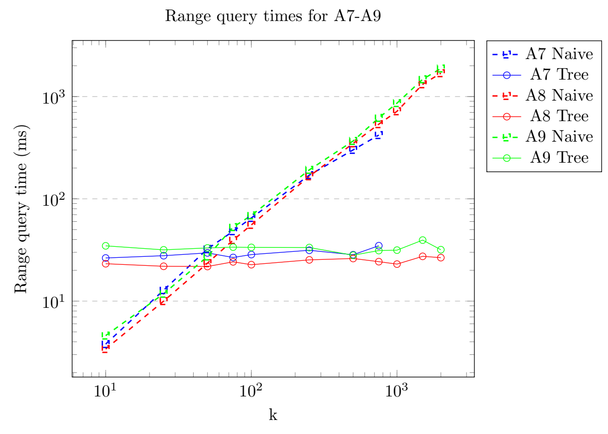
For each scenario, the average time in milliseconds over the 10 runs is given for 6 distinct operations:

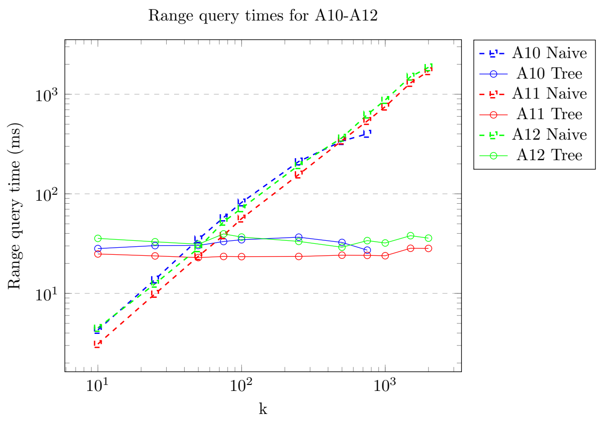
- Generation of the test data for the scenario. This includes the time spent selecting randomized points and colors, as well as the sorting of points. Note that these times are not present for the temperature data, as this is not generated at runtime.
- The building of the tree datastructure for range 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.
- 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.

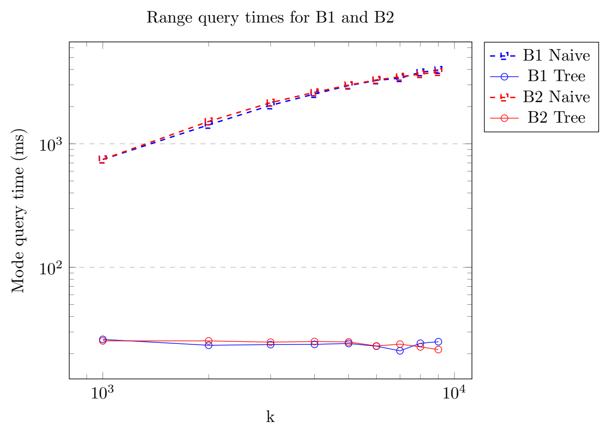
The results for range and mode queries have been compiled into graphs. Additionally, the raw tables can be found in the Appendix.



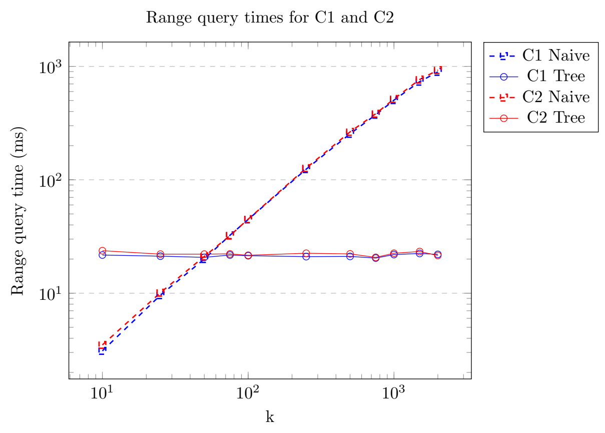




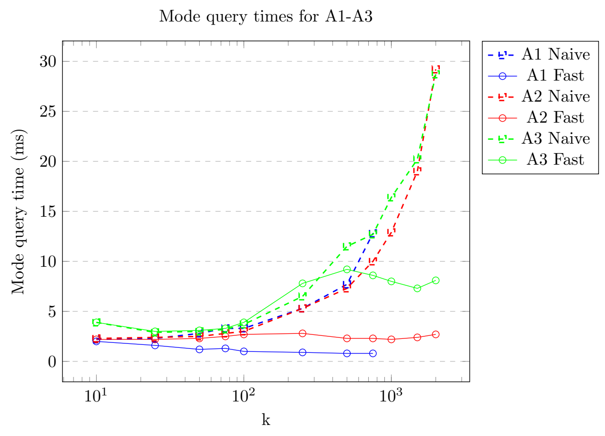




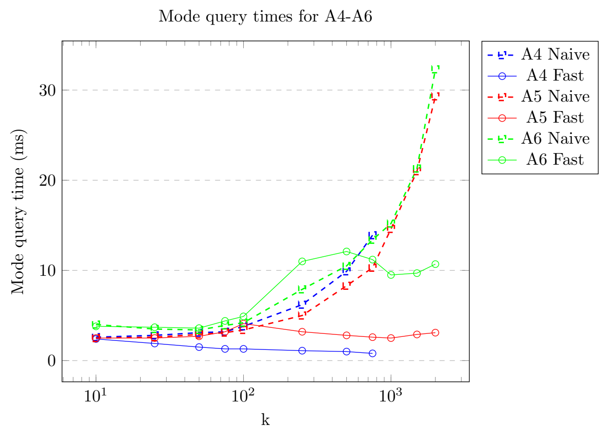
Range query times for B1 and B2
Mode query time (ms)
10^3
10^2
10^4
k
B1 Naive
B1 Tree
B2 Naive
B2 Tree



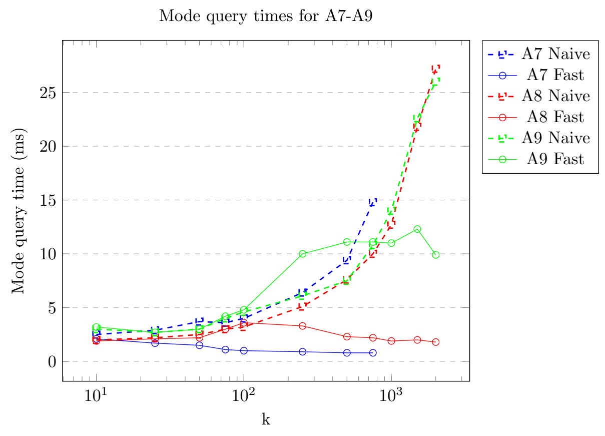
Range query times for C1 and C2
Range query time (ms)
10^3
10^2
10^1
k
C1 Naive
C1 Tree
C2 Naive
C2 Tree



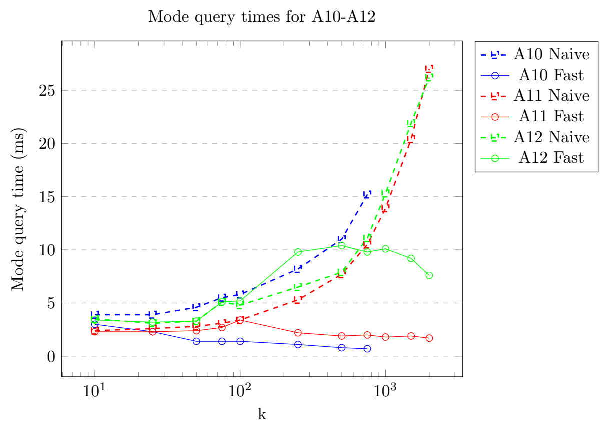
Mode query times for A1-A3
Mode query time (ms)
30
25
20
15
10
5
0
10^1
10^2
10^3
k
A1 Naive
A1 Fast
A2 Naive
A2 Fast
A3 Naive
A3 Fast



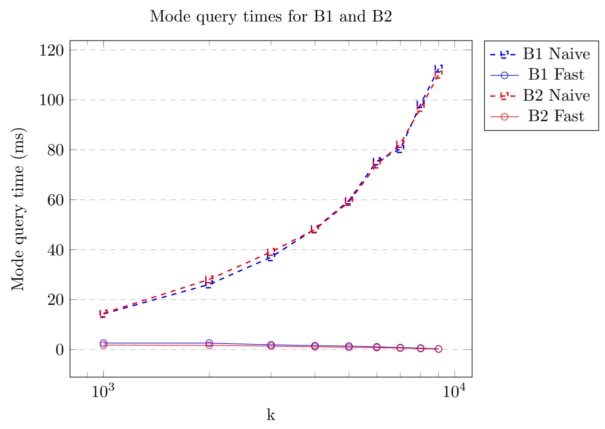
Mode query times for A4-A6
Mode query time (ms)
30
20
10
0
10^1
10^2
10^3
k
A4 Naive
A4 Fast
A5 Naive
A5 Fast
A6 Naive
A6 Fast

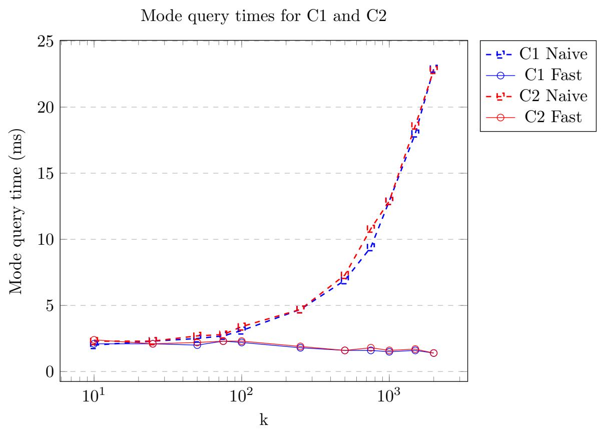


Mode query times for A7-A9
Mode query time (ms)
25
20
15
10
5
0
10^1
10^2
10^3
k
A7 Naive
A7 Fast
A8 Naive
A8 Fast
A9 Naive
A9 Fast



Mode query times for A10-A12
Mode query time (ms)
25
20
15
10
5
0
10^1
10^2
10^3
k
A10 Naive
A10 Fast
A11 Naive
A11 Fast
A12 Naive
A12 Fast





Discusssion

For range queries, the results we see are largely expected. In 1D-1-k-20-0-0 through 1D-100-k-100-150-95 we see that the naive implementation follows a roughly linear relation between k and the query time, mathcing our expectation of it being $O(\log n + k)$. In 1D-10-k-100-0-0-FRAC and 1D-10-k-100-150-95-FRAC, 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 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 $n = 100,000$.

For mode, results are also mostly in line with expectation. We do note that we seem to see a decrease in computation time for larger k values, but this is most likely due to the fact that relatively more of the window is precomputed, reducing the need to use the naive approach.

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.

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.
- $\gamma = 200$ colors are now seeded in the clustered cases instead of 150 for the large instances.
- We now have an additional parameter in r for our mode operation. This was arbitrarily set to $r = 5$ for testing of generated sets, however a more comprehensive analysis of the effect of r on the computation times is provided in the analysis of the real life data.

An overview of the tested artificial scenarios is as follows:

Uniform color scenarios				
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

Clustered 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

Real life data

As we already had a usable dataset in the form of the weather based one generated for 1D testing, this was reused when testing the 2D implementation. Additional data was however also added, as 2D datasets are widely available.

We chose to use annotated map data as our secondary source of test data. OpenStreetMap [Ope24] was used in order to download map data from various places around Utrecht, and QGIS [QGI24] was used in order to render a simplified version of the map. See 3 for an example.
This image was then sampled in order to produce colored points. The location of the point was simply 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

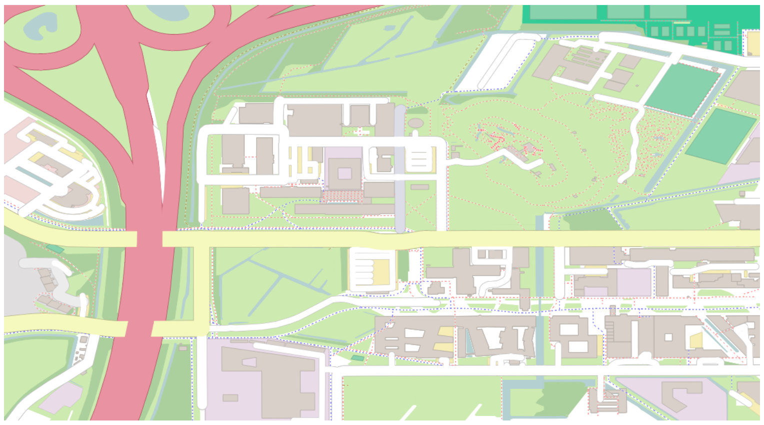


Figure 3: Simplified map view of Utrecht Science Park, centered on the Buys Ballot Building.

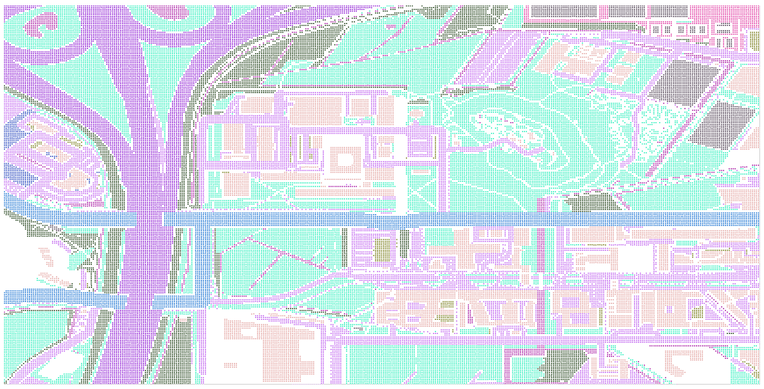
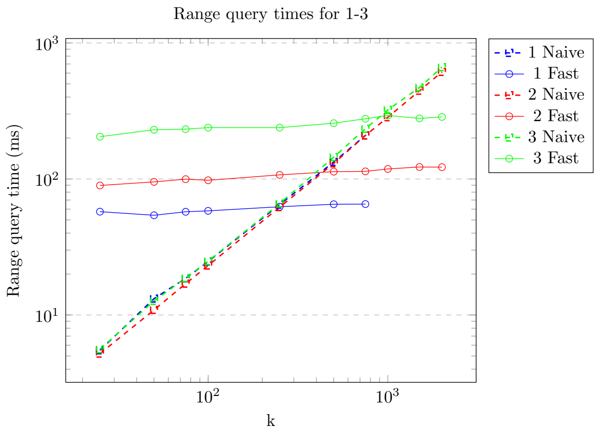


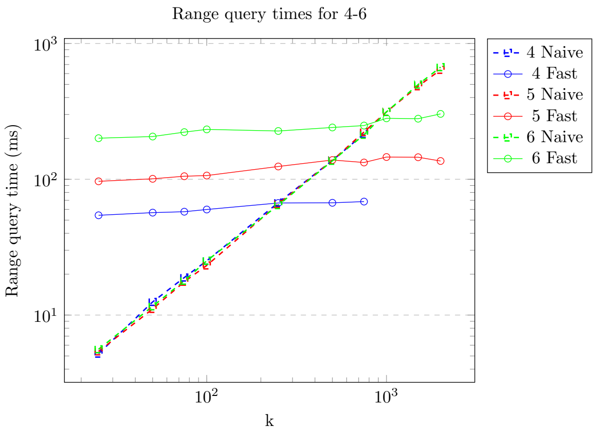
Figure 4: Point sampling taking from the simplified map view.

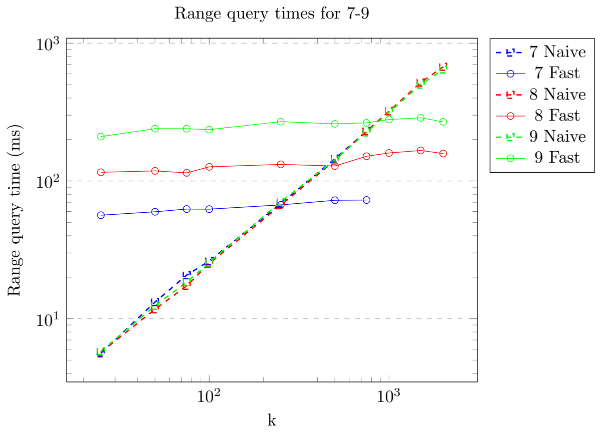
to discourage exact x and y coordinate matches for different points. Sampling rate was set such that the total instance size was around $50,000$ points for each of the included maps.

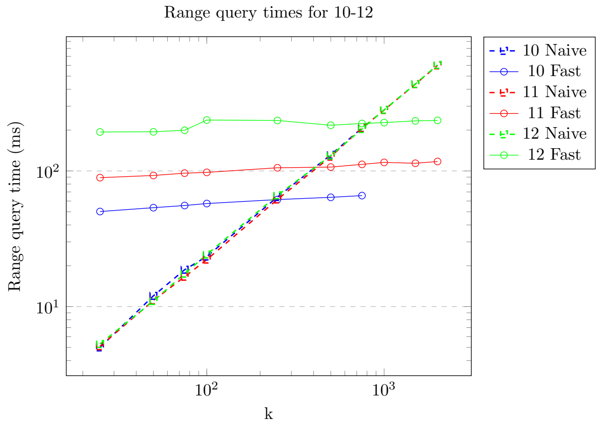
A total of 10 of these datasets were generated.

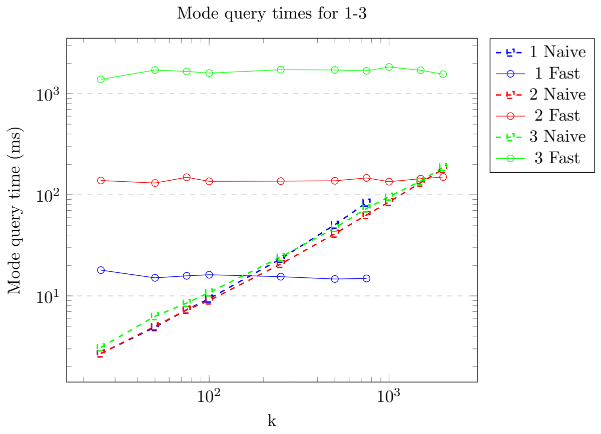
Results

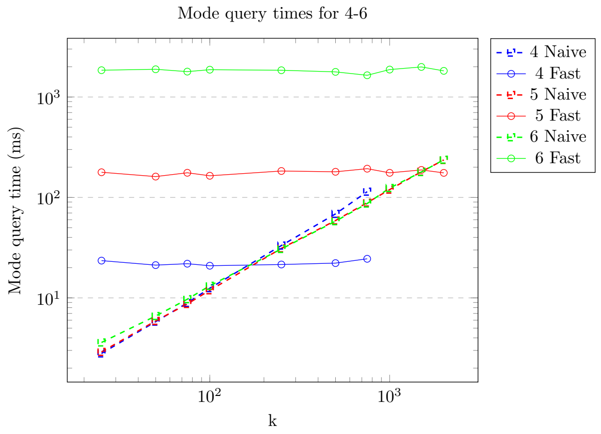

Range query times for 1-3
Range query time (ms)
k
1 Naive
1 Fast
2 Naive
2 Fast
3 Naive
3 Fast

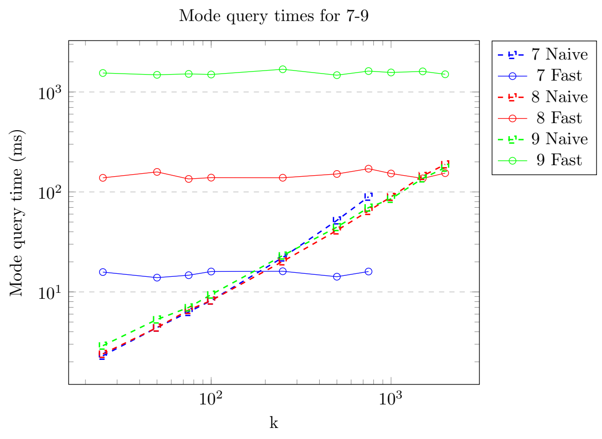
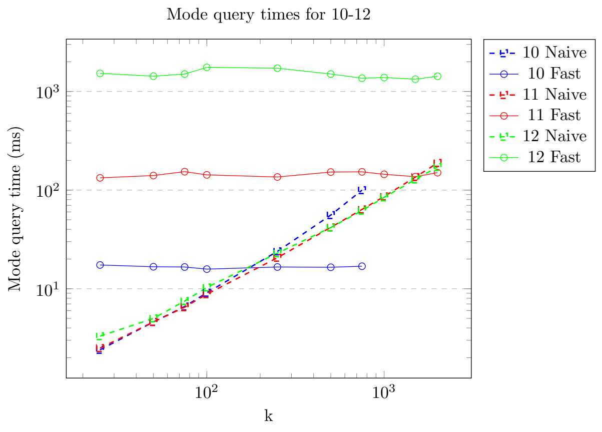
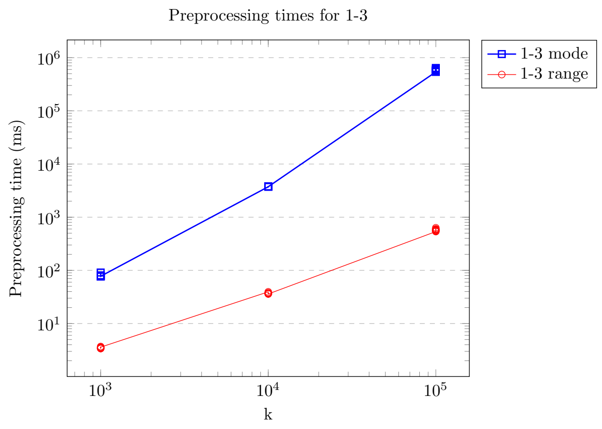
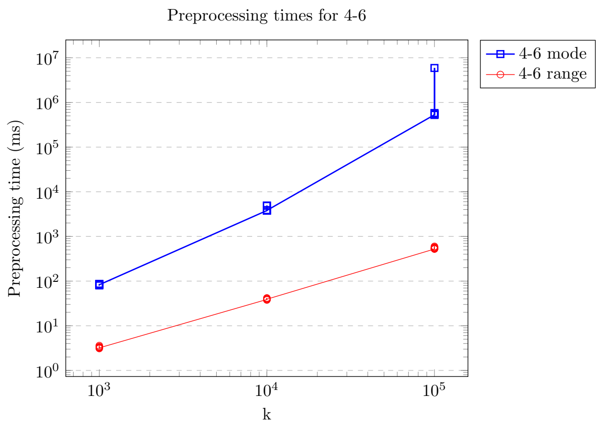
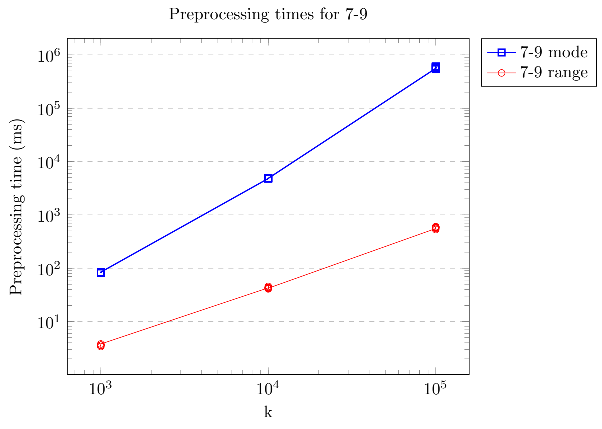
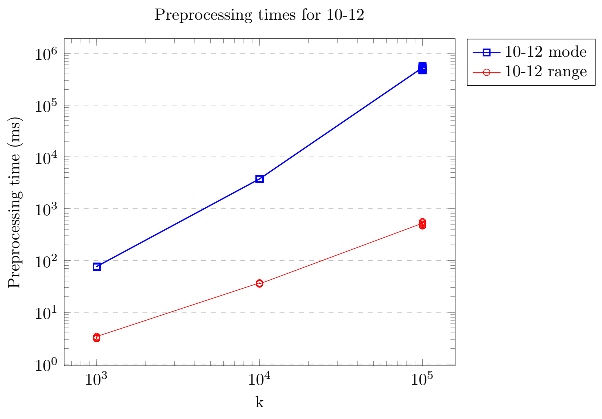

Range query times for 4-6
Range query time (ms)
k
4 Naive
4 Fast
5 Naive
5 Fast
6 Naive
6 Fast


Range query times for 7-9
Range query time (ms)
k
7 Naive
7 Fast
8 Naive
8 Fast
9 Naive
9 Fast


Range query times for 10-12
Range query time (ms)
k
10 Naive
10 Fast
11 Naive
11 Fast
12 Naive
12 Fast


Mode query times for 1-3
Mode query time (ms)
k
1 Naive
1 Fast
2 Naive
2 Fast
3 Naive
3 Fast


Mode query times for 4-6
Mode query time (ms)
k
4 Naive
4 Fast
5 Naive
5 Fast
6 Naive
6 Fast


Mode query times for 7-9
Mode query time (ms)
k
7 Naive
7 Fast
8 Naive
8 Fast
9 Naive
9 Fast

Mode query times for 10-12
Mode query time (ms)
k
10 Naive
10 Fast
11 Naive
11 Fast
12 Naive
12 Fast

Preprocessing times for 1-3
Preprocessing time (ms)
k
1-3 mode
1-3 range

Preprocessing times for 4-6
Preprocessing time (ms)
k
4-6 mode
4-6 range

Preprocessing times for 7-9
Preprocessing time (ms)
k
7-9 mode
7-9 range

Preprocessing times for 10-12
Preprocessing time (ms)
k
10-12 mode
10-12 range

References

[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.

[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.