2D Orthogonal Range Search

Mads Ravn, 20071580

Master's Thesis, Computer Science
May 2015
Advisor: Kasper Green Larsen

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

▶in English. . . ◀

Resumé

▶in Danish. . . ◀

Acknowledgements

▶. . . ◀

Mads Ravn,
Aarhus, May 10, 2015.

Contents

Abstract **iii**

Resumé **v**

Acknowledgments **vii**

1 Introduction **3**

I Theory **7**

2 Related Work **9**
2.1 kd-trees 9
2.2 Range trees 12
2.3 Composite-number space 15
2.4 Summary 15

3 Primary Work **17**
3.1 Simple Range Search 17
3.1.1 Preliminaries 18
3.1.2 Solving the ball-inheritance problem 20
3.1.3 Solving range reporting 21
3.2 Original Range Search 23
3.2.1 Preliminaries 24
3.2.2 Solving range reporting 25
3.3 Summary 27

II Practical **29**

4 Implementation **31**
4.1 Language 31
4.2 Design choices 31
4.3 Ting der skal sættes ind hist og her 32

5 Analysis **35**
5.1 Vertical and horizontal slices 35
5.2 Comparision of SRS and kd-tree with a constantly small k 44

6 ►. . . ◄ **53**

7 Conclusion **55**

Primary Bibliography **55**

Chapter 1

Introduction

Orthogonal range searching is one of the most fundamental and well-studied problems in computational geometry. Even with extensive research over three decades a lot of questions remain. In this thesis we will focus on $2D$ orthogonal range searching: Given n points from $\mathbb{R}^2$ we want to insert them into a data structure which will be able to efficiently report which points lie within a given axis-aligned query rectangle $\mathbb{Q} \subseteq \mathbb{R}^2$.

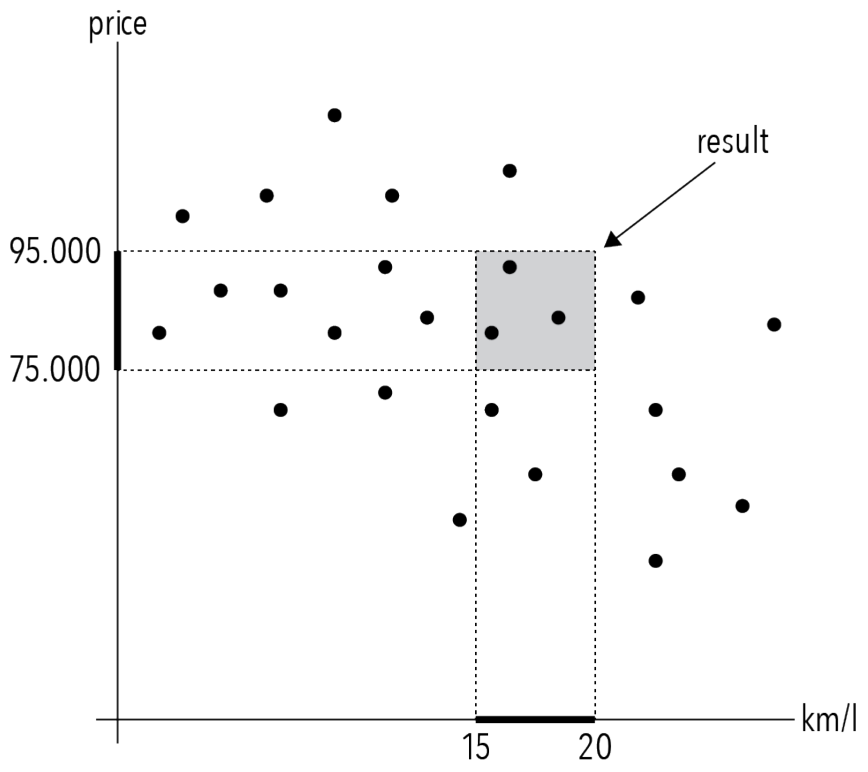

Figure 1.1: Example of an orthogonal range query

To motivate the problem, consider a database of vehicles for sale. Each vehicle has measurable attributes like price, the year the model was released, engine size, amount of doors, gasoline consumption in kilometers per liter, size and maximum speed. Perhabs a buyer is interested in finding cars which cost between $75,000$ and $95,000$ and can drive between 15 and 20 kilometers per liter of gasolin. We can see such a search on figure 1.1, where each point within the gray area represents a car which fits the criteria, i.e. a search on two parameters is equivalent to finding all points within a 2-dimensional orthogonal range query. When performing a seach, two attributes are picked and a range for both attributes can be set, giving a 2-dimensional search query. A point in the graph reprensents the ID of the car with which the car can be looked up in the database to find the other attributes of the car. We can think of each car as a point with one coordinate per attribute. Given the ranges of two attributes we want to find those of the cars in the database which lie within the search query. In the example on figure 1.1 the search range is the 2d rectangle $[15; 20] \times [75,000; 95,000]$ returning three cars as the result.

The objective of this thesis is to study a variety of *orthogonal range searching* data structures. The main focus will be to introduce the *Simple Range Search* data structure. It is a simplification of an orthogonal range searching data structure by Chan et al. [2], which will be referred to as the *Original Range Search* data structure. We are going to describe the kd-tree which will be the reference data structure in our analysis of the Simple Range Search data structure. We are going to describe the range tree which shares some of properties of the Simple Range Search and Original Range Search data structures.

We show that a range query to the Simple Range Search data structure has a faster worst-case running time than a range query to the kd-tree. We also show that the Simple Range Search data structure is able to compete with the kd-tree, and even strongly outperform the kd-tree in cases where the shape of the query is a long thin slice through the attribute area. **►Andet ord◄**

The model of computation used in this thesis is the w-bit word-RAM model by Fredman and Willard [4]. In the word-RAM model of computation, the memory is divided into words of w bits. Given a set P of n points with coordinates from a universe U, a word will have enough bits to store the integer address of any index into P and enough bits to store any element from U. Thus, $w = \Omega(\lg n)$ and $w = \Omega(\lg U)$. Under the word-RAM model all standard word operations take constant time. This includes standard word operation from modern programming languages such as integer addition, subtraction, multiplication, division, shifts and the bit-wise operators AND, OR and XOR. Reading a single word from memory or writing a single word to memory also takes constant time. The number of bits in a word is found by the largest element which has to fit into a word. This means that it is often possible to divide the word into smaller logical blocks which can fit more than one integer.

Outline. Has yet to be written.

Notation. The set of integers $\{i, i+1, \ldots, j-1, j\}$ is denoted by $[i, j]$. When no base is explicitely given logarithm will have base 2. ϵ is an arbitrary small constant greater than 0. Given an array A, $A[i]$ denotes the entry with index

i in A and $A[i, j]$ denotes the subarray containing the entries from i to j in A, including both $A[i]$ and $A[j]$. $A[1..n]$ denotes an array A of size n with entries 1 to n. Throughout the thesis the successor of x in a set will be meant as the smallest number which is greater or equal to x in that set - symmetrically, the same applies for predecessor of x which is the biggest number less or equal to x. The work will be done under the assumption that no two points will have the same x-coordinate and no two points will have the same y-coordinate. This is a unrealistic assumption in practice, but it can easily be remedied by having the points lie in a *composite-number space* since we only need a total ordering of our points.

Part I

Theory

Chapter 2

Related Work

This chapter will describe two well-known data structures for static range searching: the kd-tree and the range tree. The kd-tree is the current de facto standard for static range searching because of its low space complexity. It has a space complexity of $\mathcal{O}(n)$ and a running time of $\mathcal{O}(\sqrt{n} + k)$ where k is the number of results reported. In practise it uses the exact same amount of words as it holds elements[1]. The range tree uses $\mathcal{O}(n \lg n)$ words of space and has a running time of $\mathcal{O}(\lg^2 n + k)$ without *fractional cascading* and a running time of $\mathcal{O}(\lg n + k)$ with fractional cascading. The difference between the space complexities of the kd-tree and the range tree is a factor $\mathcal{O}(\lg n)$ which can become an issue when dealing with very large datasets and a limited amount of main memory. This factor can grow big for large datasets and is the reason why kd-trees are prefered in practise.

The kd-tree will be used as a reference in the comparison against the Simple Range Search data structure since they have the same space complexity. This property makes the Simple Range Search data structure quite attractive. The range tree will be used to show how much the running time can be decreased by increasing the space complexity by a factor of $\mathcal{O}(\lg n)$. The range tree will also be used to introduce some of the ideas behind the Simple Range Search and Original Range Search data structures, which is where Chan et al. [2] drew some of their inspiration.

The query time of the main data structures in this chapter are all *output-sensitive*, meaning that their running time depends on the amount of results found. The data structures themselves are static: After the initial construction of the data structures they will not be altered by insertions or deletions. **►more◄**

2.1 kd-trees

The current standard of range reporting using linear space is the kd-tree. This data structure will be used as a reference point when evaluting the results of the primary work of the thesis. With linear space it is a fitting data structure for range reporting on the RAM, and a practical solution. The kd-tree with n points can be represented as an array $A[1..n]$.**►Beskriver det fint nok at**

størrelsen er n?◀

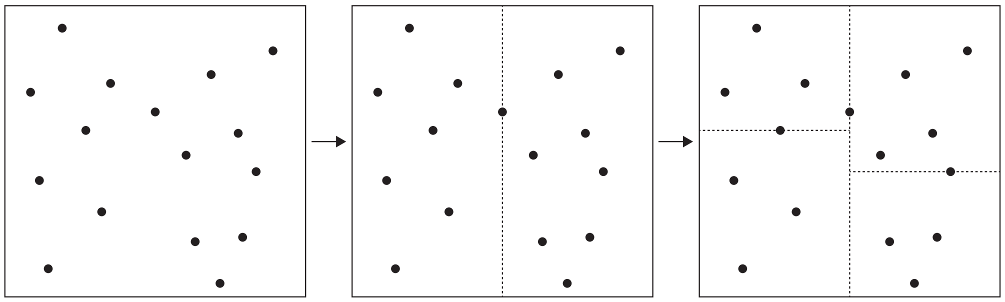

Figure 2.1: Showing the subdivision of points in a node: First dividing the points by the x-median, and then by the y-median

A kd-tree is constructed recursively: Given n points, the median of the points with respect to x is found. All points which has an x-coordinate larger than the median goes to the right child, while points which has an x-coordinate smaller than the median, and the median point, goes to the left child. At the next level the points of each node will be divided in a similar fashion, this time using the y-median and the y-coordinates instead. This is shown on figure 2.1. When dividing n points, the median will be chosen as the $\lceil n/2 \rceil$-th smallest number. Therefore a node will contain the line dividing the points given to its left child from the points given to its right child. Alternating between focussing on the x-coordinates or the y-coordinates at each level, the points are divided until only one point remains in a node. This node will then be a leaf containing that point. Thus, we end up with n leaves. This data structure uses $\mathcal{O}(n)$ words of space.

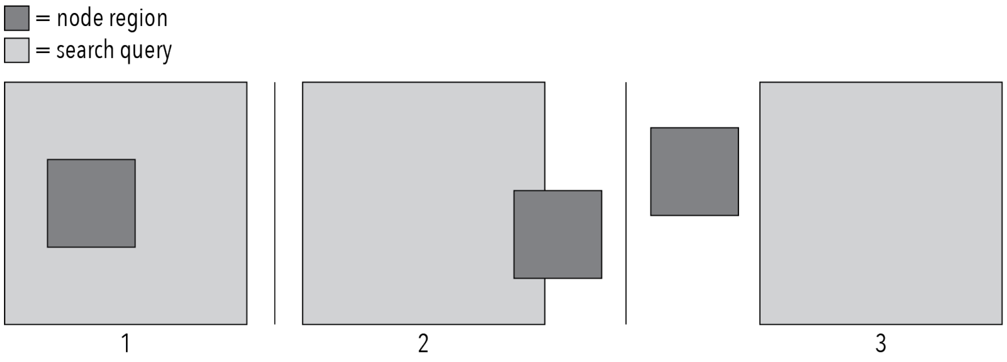

Figure 2.2: The three different situations which can occur between a search query and the region of a node

In order to search in this tree, we introduce the term *region*. The region of a node v is the smallest axis-aligned area which contains all the points in the subtree of v. The root contains all points and has the biggest region. Since each node contains a line dividing its points between both of its children, we can use this line to decrease the size of the regions of both children. Doing this halves the amount of points lying within each child region. As shown on figure 2.2, given an axis-aligned rectangular search query $q = [x_1, x_2] \times [y_1, y_2]$ and a node in the kd-tree one of three things can happen:

1. The region of the node can be fully contained in the search query, in which case the entire node and the points contained in its subtree are returned as part of the result.

2. The search query overlaps, but does not fully contain, the region of the node v. In this case the search will continue down those children of v whos region overlaps with the search query.

3. Finally the region of the node and the search query can have nothing in common in which case the search at that node stops. This check will be done by the parent of the node before recursing into the node.

►Indtil nu - skrives der så rigtigt? Er alt konsistent omkring brug af "this node" og "the children of the node" og region?◄
If a leaf is visited in the search, the point stored in the leaf is reported as part of the result if it lies within the search query. Also note that internal nodes of the kd-tree are the root of a smaller kd-tree.

Given a node, the time to report back the points stored in the subtree of that node is linear in the number of points reported. Thus, it takes $\mathcal{O}(k_v)$ time to report back all the k_v points stored in the subtree of a node v which is fully contained in the search region.

From figure 2.2 case 1 takes $\mathcal{O}(k_v)$ time and case 3 takes constant time. In order to find the time of a search query to the kd-tree, we need to obtain a bound on the time of case 2. We need to obtain a bound on the amount of nodes visited which is not fully contained in the search region. These are the nodes where an edge of the search query passes through their regions. Consider a search query where one of the edges passes through the region of the root. This edge can be thought of as infinitely long. Without loss of generality, we pick it to be a horizontal line, as seen on figure 2.3.

We want to obtain a bound on the amount of regions this edge can pass through. Let $Q(n)$ describe the amount of regions the horizontal line intersects. In order to find the amount of regions intersected by the line, we need to recall how the kd-tree is built. First a region is split in one dimension and then in the other dimension, resulting in four regions every two levels of the kd-tree constructed **►Rephrase sentence - argument should be obvious◄**. The horizontal line will intersect two of these regions. The horizontal line will also intersect the region of the root and one of the two children of the root. The running time of a query to the kd-tree with n points can be described by the recurrence:

$$Q(n) = \begin{cases} \mathcal{O}(1), & \text{if } n = 1, \\ 2 + 2Q(n/4), & \text{if } n > 1 \end{cases}$$

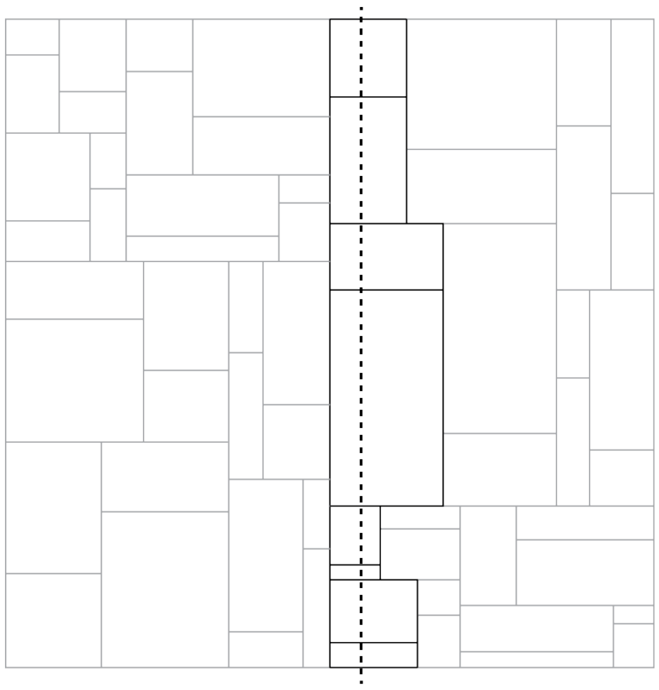

Figure 2.3: Example of a horizontal edge through the entire region of the root node

Solving this recurrence gives the solution $Q(n) = \mathcal{O}(\sqrt{n})$. A horizontal line through the region of the root in a kd-tree will intersect $\mathcal{O}(\sqrt{n})$ regions of the kd-tree, thus bounding the amount of regions of nodes a horizontal line can pass through. The exact same argument can be made for a vertical line.

Searching the kd-tree takes $\mathcal{O}(\sqrt{n} + k)$ time to report k points as a result. When the amount of points reported back as result of the search query is low, the query time per point is relatively high. Another thing to notice is that there is no time penalty per point reported. Just searching through the data structure costs $\mathcal{O}(\sqrt{n})$ time, but the time to report back points is linear in the number of points reported.

2.2 Range trees

The range tree is a data structure which supports range queries. The space complexity of this data structure is $\mathcal{O}(n \lg n)$. A normal query in a range tree uses $\mathcal{O}(\lg^2 n + k)$ time to report k points. This time can be optimized to $\mathcal{O}(\lg n + k)$ without changing the space complexity using *fractional cascading*. We will first look at how the data structure is built and how it is used for range reporting. Then we will introduce fractional cascading and see how that will change the query time. With a space complexity of $\mathcal{O}(n \lg n)$ words this data structure is not going to replace the kd-tree. Instead the range tree will serve as a way to introduce some of the ideas behind the SRS and ORS data structures.

Consider a balanced binary search tree with n keys for a 1-dimensional query on the x-coordinates. In order to answer the query $q = [x_1, x_2]$ the following is done: From the root, travel to the *least common ancestor* of x_1 and x_2. This is the node whose subtree contains both x_1 and x_2, and x_1 lies in the left subtree, while x_2 lies in the right subtree. From the least common ancestor, travel to both x_1 and x_2. While traveling to x_1, the first step is the left child of the lowest common ancestor x_1 and x_2. From here, everytime a left child is chosen as the next step in the path, the subtree in the right child will only contain points between x_1 and x_2. This entire subtree is reported back as results. Symmetrically, the same is done with the path to x_2. When a right child is chosen as the next step, the subtree in the left child is reported back as results. In a 1-dimensional search, when a node is the root of a subtree which only contains points in the search range, the node is said to be *fully contained.*

A balanced binary search tree has a space complexity of $\mathcal{O}(n)$. Reporting back the points stored in a subtree requires time linear to the amount of points in the subtree. Travelling from the root to x_1 and x_2 requires $\mathcal{O}(\lg n)$ time. Hence, the query time of a 1-dimensional search query is $\mathcal{O}(\lg n + k)$.

Range reporting in a 2-dimensional space on the kd-tree is done by using 1-dimensional sub-queries. The kd-tree alternates in which dimension the search is performed. The range tree also searches by using 1-dimensional sub-queries, but instead of alternating between dimensions, it separates them. Given a search query $q = [x_1, x_2] \times [y_1, y_2]$, it will first find the points lying in the range of $[x_1, x_2]$. Among those points, it will find the points lying in the range of $[y_1, y_2]$. This leaves us with all the points lying in the search query.

Doing the first 1-dimensional search is exactly what is accomplished using a balanced binary search tree. A balanced binary search tree is built to support range search on the x-axis of all of the points. We will call this tree the primary tree. Then for each internal node in the primary tree a new balanced binary search tree is built on all points in the leaves of the subtree of that node. We call these balanced binary search trees for auxiliary trees. The primary tree holds pointers to the auxillary tree for each node.

A range query $q = [x_1, x_2] \times [y_1, y_2]$ on the range tree is answered in the following way. From the least common ancestor of x_1 and x_2, the search travels down to x_1 and x_2. On the way to x_1 and x_2, each node that is fully contained in $[x_1, x_2]$ will be flagged. Using the auxilliry tree of each node that is flagged, a search will be done to find the points in the range $[y_1, y_2]$.

The height of a balanced binary search tree containing n points is $\lg n$. Each point p in the primary tree is only stored in the auxillary trees of nodes on the path to the leaf containing the point p. This means that each point p is only stored once per level in the primary tree. Each auxillary tree uses space linear to the amount of points it holds. Thus, the space complexity of a range tree is bounded by $\mathcal{O}(n \lg n)$.

The query time for each auxillary tree that is searched is $\mathcal{O}(\lg n + k_v)$, where k_v is the amount of points that is reported back by the auxillary tree at the node v in the primary tree. The amount of auxillary trees which will be searched is bounded by the length of the path from the least common ancestor of x_1 and x_2 to the leaves containing x_1 and x_2. This path can at most visit two nodes per level of the primary tree, and the length is thus bounded by $\mathcal{O}(\lg n)$. The query time of a range search in the range tree is then

$$\sum_v \mathcal{O}(\lg n + k_v) = \mathcal{O}(\lg^2 n + k)$$

where v are the nodes flagged on the path to x_1 and x_2 from their least common ancestor.

Fractional cascasding can be used to speed up the query time without changing the space complexity of the data structure. Instead of using a balanced binary search tree as the auxillary data structure, we are going to use an array. This array will contain the same points as the auxillary balanced binary search tree did. The points in the array will be sorted by their y-coordinate. At the node v, each entry in the array $A_v$will contain a point and two pointers. One pointer will be pointing to an entry in the auxillary array of the left child of v, while the other pointer will be pointing to an entry in the auxillary array of the rigth child of v. We call these the left pointer and the right pointer, respectively. Suppose that $A_v[i]$ stores a point p. Then the left pointer at $A_v[i]$ will be pointing to the first entry in the left childs auxillary array containing a point with a y-coordinate greater or equal to p_y. The same applies to the right pointer of $A_v[i]$, pointing to the right child instead of the left child.

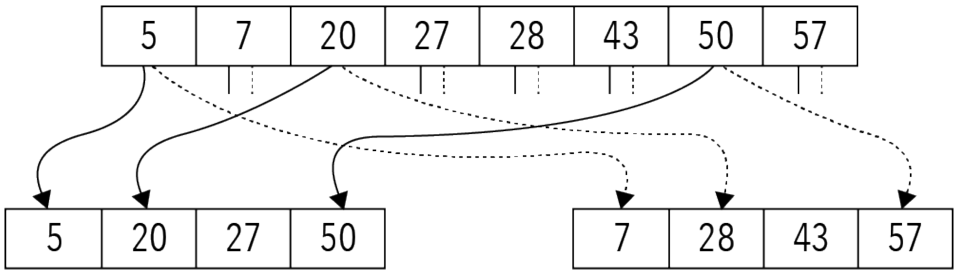

Figure 2.4: Example of an fractional cascading

Searching the range tree with fractional cascasding starts by finding the least common ancestor of x_1 and x_2. At this node, a binary search is done in order to find the first entry in the auxillary array which y-coordinate is greater or equal to y_1. At any given node, we call the position of this entry τ. We walk from the least common ancestor of x_1 and x_2 to both x_1 and x_2, finding all the nodes which are fully contained in $[x_1, x_2]$. Each time a left child is visited on the path to x_1 or x_2, the left pointer is used to update τ. The entry at position τ_v is the first element in A_v which is greater or equal to y_1. Finding the index of this element at a child-node is a constant-time operation using the left pointer. Symmetrically, when a right child is visited on the path to x_1 and x_2, the position of τ is updated using the right pointer. This can be seen on figure 2.4 When a fully contained node is found, we look in the auxillary array from the position of τ and k_v entries forward in order to report k_v points back as result. This is done by incrementing the position of τ until the point at that entry is no longer within the range of $[y_1, y_2]$. This takes $\mathcal{O}(1 + k_v)$ time. The total query time now becomes

$$\sum_v \mathcal{O}(1 + k_v) = \mathcal{O}(\lg n + k)$$

where v are the nodes flagged on the path to x_1 and x_2 from their least common ancestor.

2.3 Composite-number space

In order to ensure all points have unique x-coordinates and unique y-coordinates, the points are translated into *composite-number space*[1]. A composite number of two numbers x and y is denoted by $(x \mid y)$. A total ordering on the composite-number space is defined by using lexicographic order. Given two composite numbers $(x_1 \mid y_1)$ and $(x_2 \mid y_2)$, we define the order as

$$(x_1 \mid y_1) < (x_2 \mid y_2) \iff x_1 < x_2 \text{ or } (x_1 = x_2 \text{ and } y_1 < y_2)$$

Given a set P of n distinct points from $\mathbb{R}^2$, we translate each point $(x, y) \in P$ into composite-number space by assigning the point new set of coordinates: $(x, y) := ((x \mid y), (y \mid x))$. No two points will have the same x-coordinate unless the points are identical. The same holds for the y-coordinate.
In order to perform a range query $q = [x_1, x_2] \times [y_1, y_2]$ in composite-number space, the query will have to be transformed. This transformed range query will be $\hat{q} = [(x_1 \mid -\infty), (x_2 \mid +\infty)] \times [(y_1 \mid -\infty), (y_2 \mid +\infty)]$. It follows that

$$(x, y) \in q \iff ((x \mid y), (y \mid x)) \in \hat{q}$$

2.4 Summary

The kd-tree is a data structure using $\mathcal{O}(n)$ words of space and supports a range query in $\mathcal{O}(\sqrt{n} + k)$ time. It is built by continually subdividing smaller and smaller regions of the tree until a region only contains one point. A range search query will then match the query region to the region of node to see if there is any overlap or full containment.

The range tree is a data structure using $\mathcal{O}(n \lg n)$ words of space and support a range query in $\mathcal{O}(\lg n + k)$ time. It is built by creating a tree with n leaves and dividing the points to the leaves, such that all the leaves to the left of a leaf contain points with a smaller x-coordinate than the point at the leaf. All the internal nodes of the tree contain an auxillary tree which has the same property just with the y-coordinate of the points contained in the subtree. This property allows a search query to quickly locate the subtrees containing only points between $[x_1, x_2]$ and $[y_1, y_2]$.

The $\mathcal{O}(\lg n + k)$ running time of the range tree is faster than the $\mathcal{O}(\sqrt{n} + k)$ running time of the kd-tree. However, the $\mathcal{O}(\sqrt{n})$ part is based on a rather pessimistic idea that a range query will overlap, but not fully include, a lot of regions stretching over the two extrema in one dimension.

Chapter 3

Primary Work

This chapter will introduce the Simple Range Search and Original Range Search data structures. The Original Range Search data structure by Chan et al. [2] is a tree-like data structure with auxillary data structures. It has a space complexity of $\mathcal{O}(n)$ and a supports search queries in $\mathcal{O}(\lg\lg n + (1+k)\cdot\lg^{\epsilon} n)$ time, where k is the amount of results reported and ϵ is an arbitrarily small constant greater than 0.

The Simple Range Search data structure is a simplification of the Original Range Search and therefore they have the same underlying data structure. The Simple Range Search data structure has some different auxillary data structures and fewer of them. The data structure has a space complexity of $\mathcal{O}(n)$ and supports search queries in $\mathcal{O}(\lg n + (1+k)\lg^{\epsilon} n)$ time. Going forward,*ORS* will be used as shorthand for *Original Range Search* and *SRS* will be used as shorthand for *Simple Range Search*.

The SRS data structure has a time complexity of $\mathcal{O}(\lg n + (1+k)\cdot\lg^{\epsilon} n)$ which is greather than the time complexity of the ORS data structure with $\mathcal{O}(\lg\lg n + (1+k)\cdot\lg^{\epsilon} n)$. However, the SRS data structure is far more simple - both in code and the auxilary data structures used. The difference between the running time constant in $\mathcal{O}(\lg\lg n)$ and $\mathcal{O}(\lg n)$ is far greater than the difference between $\lg\lg n$ and $\lg n$. This makes the SRS data structure faster than the ORS data structure in practice.

Each section will start with a *preliminaries* subsection. This subsection will describe some of the auxiliary data structures used in the section. As in Chapter 2, the main data structures in this chapter are output-sensitive and static.

3.1 Simple Range Search

This section will introduce the primary work of this thesis. It will show how the SRS data structure is built and how range reporting is done using the data structure. The data structure uses $\mathcal{O}(n)$ space and supports search queries in $\mathcal{O}(\lg n + (1+k)\cdot\lg^{\epsilon} n)$ time. This is the same space complexity as the kd-tree. The query time is different in that $\mathcal{O}(\lg n)$ is smaller than $\mathcal{O}(\sqrt{n})$, but there is a cost of $\mathcal{O}(\lg^{\epsilon} n)$ per point reported.

The SRS data structure relies heavily on the *ball-inheritance* data structure. The ball-inheritance structure is a tree with n labelled balls at the root. In $\lg n$ steps it will distribute the balls from the root of the tree to n leaves in the tree. Solving the ball-inheritance problem is to follow a ball from any given node to its leaf. We formally define the problem below **►ref?◄**. Succint rank queries are an important part of this chapter, playing a key role in solving the *ball-inheritance problem*.

3.1.1 Preliminaries

Rank Space Reduction

Given n points from a universe U, the rank of a given point in a sorted list of points is defined as the amount of points which preceed it in the list. Given two points $a, b \in U : a < b$ *iff* $rank(a) < rank(b)$. Expanding this concept to 2 dimensions we have a set P of n points on a $U \times U$ grid. We compute the *x-rank* r_x for each point in P by finding the rank of the x-coordinate among all the x-coordinates in P. The *y-rank* r_y finds the rank of y-coordinate among all of the y-coordinates in P. Using *rank space reduction* on P, a new set P^* is constructed where $(x, y) \in P$ is replaced by $(r_x(x), r_y(y)) \in P^*$. Given a range query $q = [x_1, x_2] \times [y_1, y_2]$, a point $(x, y) \in P$ is found within q *iff* $(r_x(x), r_y(y))$ is found within $q^* = [r_x(x_1), r_x(x_2)] \times [r_y(y_1), r_y(y_2)]$. Computing the set P^* from P using rank space reduction, P^* is said to be in rank space. While the n points could be represented by $\lg U$ bits in P, they can now be represented by $\lg n$ bits in P^* with $\lg n \ll \lg U$ when $n \ll U$ which saves memory. Given n points from U we can translate them to rank space by inserting them in an array $A[1..n]$ and sort A. A point's index in A is now its rank. In order to translate a search query $q = [x_1, x_2]$ to rank space, we will look up the successor of x_1 in A and the predecessor of x_2 in A. The indices of these two points delimits the search query in rank space. The array A acts as a mapping to and from rank space. **►rephrase◄** When a rank space reduction has been applied and there exists an array $A[1..n]$ mapping the elements to and from rank space, the algorithms used will only use RAM operations on integers of $\mathcal{O}(\lg n)$ bits.

Predecessor search using binary search

In order to find the rank space successor or rank space predecessor of a point, a binary search is used on a sorted array of points. This data structure uses $\mathcal{O}(n)$ space and have a query time of $\mathcal{O}(\lg n)$. By locating the first key in the array that is greater than or equal to the search query, the index of that key is the *rank space successor*. Similarly, by locating the last key that is smaller in the array, the index of that key is the *rank space predecessor*.

Succint rank queries

Consider an array $A[1..n]$ with elements from some alphabet Σ. Given an index i in the array, we can report how many elements in $A[1..i]$ are equal to $A[i]$.

This is called a *rank query*. We want to be able to compute a rank query in constant time using a data structure of $\mathcal{O}(n \lg \Sigma)$ bits. In order to do this, checkpoints are created. For each character in the alphabet Σ, the checkpoint contains the number of times that character appears in $A[1..i]$, where i is the checkpoint location. Such a checkpoint takes up $\mathcal{O}(\Sigma \lg n)$ bits of space. By placing the checkpoints $\Sigma \lg n$ entries apart of each other, all of the checkpoints use $\mathcal{O}(\frac{n}{\Sigma \lg n} \cdot \Sigma \lg n) = \mathcal{O}(n)$ bits of space.

Each entry in A contains a character from the alphabet Σ. At $A[i]$ we also store the amount of times the character at $A[i]$ occurs since the last checkpoint. This is a smaller number and can be stored using $\mathcal{O}(\lg(\Sigma \lg n))$ bits per entry in A. This is because we only need $\lg x$ bits to store a number which has a maximum value of $x - 1$. This approach fits the required space bound if $\Sigma \geq \sqrt{\lg n}$ **▶replace $\sqrt{\lg n}$ with $\lg^\epsilon n$◀**, because there Σ will dominate the complexity. Hence, storing n entries uses $\mathcal{O}(n \cdot \lg(\Sigma \lg n)) = \mathcal{O}(n \cdot \lg \Sigma) = \mathcal{O}(n)$ bits **▶Inkluderer "per level"?◀**.**▶Er det ved "to dominate" betyder i dette tilfælde?◀** Working under the word-RAM model of computation, we are able to pack the integers into words of $\lg m$ bits, where m is the maximum number which needs to be stored.

For smaller alphabets, another scheme is used. Checkpoints are still stored at every $\Sigma \lg n$ positions. These are now called major checkpoints. In addition to this, minor checkpoints are added. These minor checkpoints are added at every $\lg \lg n$ positions and contain the amount of times each character is seen since the last major checkpoint. These minor checkpoints take up $\mathcal{O}(\Sigma \lg \lg n)$ bits each. $A[i]$ now stores the character from the alphabet Σ and how many times that character occurs since the last minor checkpoint. In order to answer the rank query, a query to the last major checkpoint and the last minor checkpoint has to be made. Given that $\Sigma \lg \lg n \leq \sqrt{\lg n} \cdot \lg \lg n$, the array entries holding the amount of times $A[i]$ is seen since the last minor checkpoint fits into $\mathcal{O}(\sqrt{\lg n} \cdot \lg^2 \lg n)$ bits. Therefore it is possible to store these array entries in plain form, and with the help of a precomputed table of $n^{\mathcal{O}(1)}$ space we can answer rank queries between minor checkpoints in constant time. **▶Plain form?◀ ▶rephrase◀ ▶Maybe only need large alphabet + bit vector. Then can leave out complicated multilevel◀**

Ball Inheritance

Consider a perfect binary tree with n leaves and n labelled balls at the root. The balls have been distributed from the root to the leaves in $\lg n$ steps, where each node picks one of its children to inherit a given ball. Both children of a node will receive the same amount of balls. The balls at the root are contained in an ordered list, and they will retain this order at all internal nodes. Each node contains a list of which ball goes to which child. The level of a node is defined to be the height of the node from the leaves. The root has the highest level, while each node is one level smaller than its parent. The leaves are at level 0. Each level of the tree contains the same amount of balls, and at level i each node contains 2^i balls. Eventually each ball reaches a leaf of the tree and each leaf will contain exactly one ball. A ball can be identified by a node and the index of the ball in the list of that node. Given the identity of a ball at any level, it is possible to follow this ball down the tree to a leaf. The goal is to track a balls inheritance from a given node to a leaf and report the identity of the leaf. We call the identity of the leaf the *true identity* of a ball. **▶Omskriv afsnit til at reflektere at boldene allerede er fordelt◀**

3.1.2 Solving the ball-inheritance problem

Consider a perfect binary tree with n leaves. At level m, each node contains a bit vector $A_v[1..2^m]$ used to indicate which child-node a ball is inherited by: If $A_v[i]$ is 0 it means that the ball at index i in that node was inherited by the left child and 1 means that it was inherited by the right child. The identity of a ball is a node and an index into the list of that node. Given a node v and an identity of a ball, we can now calculate the ball's identity in the child node which inherits the ball. The node can answer the query $rank_v(k) = \Sigma_{i \leq k} A_v[i]$. If a ball is inherited by the right child node its new identity at that node is $rank_v(i)$ because that is how many 1's that preceed it in the current node. If a ball is inherited by the left child node the new identity is then $i - rank_v(i)$. With this information it is possible to traverse down the tree following a ball from any given node to a leaf. There are n balls per level stored in the bit vectors of the nodes on that level. Each level in the tree uses $\mathcal{O}(n)$ bits to store the bit vectors. This adds up to $\mathcal{O}(n \lg n)$ bits, or $\mathcal{O}(n)$ words in all. This trivial solution to the ball-inheritance problem uses $\mathcal{O}(\lg n)$ query time, given that it follows a ball $\mathcal{O}(\lg n)$ steps down to its leaf. The rank function is a constant time query 3.1.1. **▶Hvordan skal jeg referere til dette?◀**

Faster Queries

A bit vector is an array with entries from the alphabet $\Sigma = \{0, 1\}$, where each entry is used to indicate whether a left or right child has been chosen to inherit a given ball. By expanding the alphabet we can point to the children's children, $\Sigma = \{0, 1, 2, 3\}$, the children's children's children, $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$, and so forth. Expanding the alphabet will use $\mathcal{O}(n \lg \Sigma)$ bits per level. Storing a pointer from level i to level $i + \Delta$ increases the storage space by Δ bits per ball, but also enables the ball to be inherited by 2^Δ descendants. By expanding the alphabet the query time can be lowered since it is possible to take bigger steps down the tree. Just like with a parent-to-child step, bigger steps are supported by succint rank queries. In order to determine the identity of a ball b Σ levels below, we need to know the destation node and how many balls before b chose the same destation node. Using succint rank queries to determine the rank of a ball Σ levels down is thus a constant time query.

Using this concept, we pick B such that $2 \leq B \leq m$, where $m = \lg n$ is the height of the tree. All levels that are a multiple of B^i expand their alphabet such that the balls can also reach B^i levels down. If a target level does not exist, the ball points to its leaf. We need at most visit B levels that are multiple of B^i before reaching a level that is multiple of B^{i+1}, making it possible to jump down the tree with bigger and bigger steps.

Storing the expanded alphabets at each level that is a multiple of B^i costs B^i bits per ball. The total cost is then

$$\sum_{i=1}^{\lg_B \lg n} \frac{\lg n}{B^i} \cdot \mathcal{O}(B^i) = \mathcal{O}(\lg n \cdot \lg_B \lg n)$$

bits per ball, at all levels. With n balls, this is $\mathcal{O}(n \lg_B \lg n)$ words of space with query time of $\mathcal{O}(B \lg_B \lg n)$. If we pick a $B \leq \lg^\epsilon n$ we can reduce $\lg_B \lg n$ to $\lg_{\lg^\epsilon n} \lg n = \frac{1}{\epsilon}$. Thus, the space complexity for the ball-inheritance data structure is $\mathcal{O}(\frac{1}{\epsilon} \cdot n) = \mathcal{O}(n)$ words of space. By picking $B = \lg^{\epsilon/2} n \geq \lg \lg n$ we can find the upper bound on the query time as follows:

$$\begin{aligned}\mathcal{O}(B \lg_B \lg n) &= \mathcal{O}(B \lg \lg n) = \mathcal{O}(\lg^{\epsilon/2} n \cdot \lg \lg n) \\ &= \mathcal{O}(\lg^{\epsilon/2} n \cdot \lg^{\epsilon/2} n) = \mathcal{O}(\lg^\epsilon n)\end{aligned}$$

The ball-inheritance problem can thus be solved in $\mathcal{O}(\lg^\epsilon n)$ time using $\mathcal{O}(n)$ words of space. **►Det er fordi $\lg_B \lg n$ er det største i som beskriver $B^i \leq \lg n$ ◄**

3.1.3 Solving range reporting

Consider a perfect binary tree with n leaves. The root contains n points in 2-d rank space. The elements in the ball-inheritance structure are points, so the words *ball* and *point* can be used interchangeably. The n balls at the root of the tree are sorted by their y-rank. When distributing the balls for inheritance, a node will give both its children half of its balls: the lower half sorted by the x-rank to its left child and the upper half by x-rank to its right child. The order of the balls in a child node will be the same as the parent node. The actual coordinates of the balls are only stored at the leaves. This is how the ball-inheritance data structure was described in the previous section. The ball distribution has been specified. With this distribution, some facts about the tree can be stated. We know that the x-coordinates of the balls in the leaves are sorted from left to right - smallest to highest. The way the balls are distributed from the root, the x-coordinates are responsible for the inheritance path. **►rephrase◄** Because the nodes are sorted by their y-rank in the root node and that they keep this order, the balls in a node list at any given node is ordered by their y-rank. These two facts will be used to solve the range reporting.
Since the actual coordinates of the points are only stored once, this data structure uses linear space.

Given a range query $q = [x_1, x_2] \times [y_1, y_2]$ the rank successors of x_1 and y_1 and the rank predecessors of x_2 and y_2 are looked up. We know from section **►ref◄** that a range query can be translated to a rank space query. We call these $\hat{x}_1, \hat{y}_1, \hat{x}_2$ and $\hat{y}_2$. We now have our query q in rank space: $\hat{q} = [\hat{x_1}, \hat{x_2}] \times [\hat{y_1}, \hat{y_2}]$. We use $\hat{x}_1$ and $\hat{x}_2$ to find the lowest common ancestor, $LCA(\hat{x}_1, \hat{x}_2)$. This node's subtree contains at least all the points with an x-coordinate between x_1

and x_2.

At the root we mark the positions of $\hat{y}_1$ and $\hat{y}_2$ on the bit vector. This range indicates which balls lie within the range of $[y_1, y_2]$. i_v and j_v will denote this range in the bit vector of the node v. When searching for points lying within this range, a node will update this range to fit its children. The updated range at the left child l will be $i_l = i_v - rank_v(i_v)$ and $j_l = j_v - rank_v(j_v)$. The updated range at the right child r will be $i_r = rank_v(i_v)$ and $j_r = rank_v(j_v)$. This is the same way the rank query was used in section 3.1.2. Now instead of just following a given ball, we keep track of a range of balls. This concept is seen on figure 3.1

Traversing from the root to the LCA, this y-range will be updated accordingly. We know the positions of the leaves containing $\hat{x}_1$ and $\hat{x}_2$ so we can traverse from the LCA down to each of them. Traversing to $\hat{x}_1$, the first stop is the left child of the LCA. From here, each time a node selects its left child as the path to $\hat{x}_1$ we know that the subtree contained in the right child only contains points with x-coordinates between x_1 and x_2. Symmetrically, the same applies when going right from the LCA: Each time a node selects a right child on the path to $\hat{x}_2$ the subtree contained in the left child only contains points between x_1 and x_2. Such a subtree is said to be fully contained. This concept is seen on figure 3.2.

Each time a fully contained subtree is found, we want to follow all the balls lying in the y-range of the root of the subtree to their leaves. This is exactly what the ball-inheritance solves: We are given a list of ball identities and want to find their actualy coordinates. Finding a ball's coordinate from here takes $\mathcal{O}(\lg^{\epsilon} n)$ time per ball.

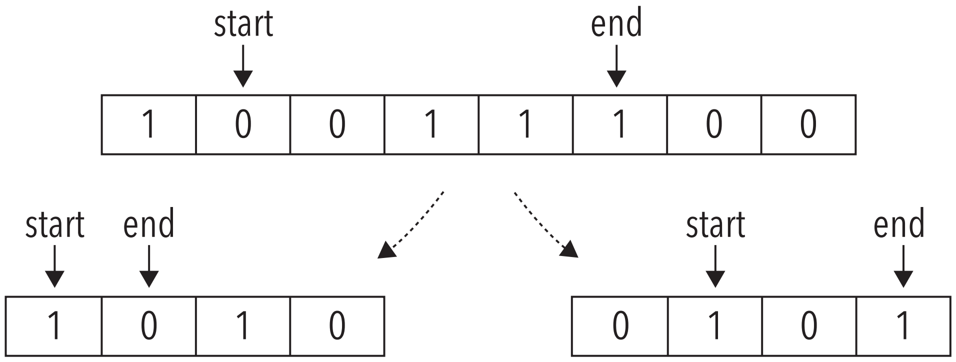

Figure 3.1: Example of nodes inheriting their bit vector ranges from their parent.

The actual coordinates of the points are only stored at the leaves which then takes up $\mathcal{O}(n)$ words of space. The rest of the tree contains $\lg n$ levels of bit vectors of n bits taking $\mathcal{O}(n \lg n)$ bits, $\mathcal{O}(n)$ words. Looking up the rank-space predecessor and successor of x_1, x_2, y_1 and y_2 using a simple binary search at the root requires $\mathcal{O}(n)$ space and $\mathcal{O}(\lg n)$ time. Summing it up, the entire data structure uses $\mathcal{O}(n)$ words of space.

Walking from the root to the LCA requires $\mathcal{O}(\lg n)$ steps. Visiting $\hat{x}_1$ and $\hat{x}_2$ requires $\mathcal{O}(\lg n)$ steps each. Visiting each of the k leaves in the subtrees between $\hat{x}_1$ and $\hat{x}_2$,containing the points which will be reported as a result, takes $\mathcal{O}(k \cdot \lg^{\epsilon} n)$ time.

This adds up to $\mathcal{O}(\lg n + (1 + k) \cdot \lg^{\epsilon} n)$ query time to report k points as results. **►Tilføj en sætning eller to - just remake this◄**

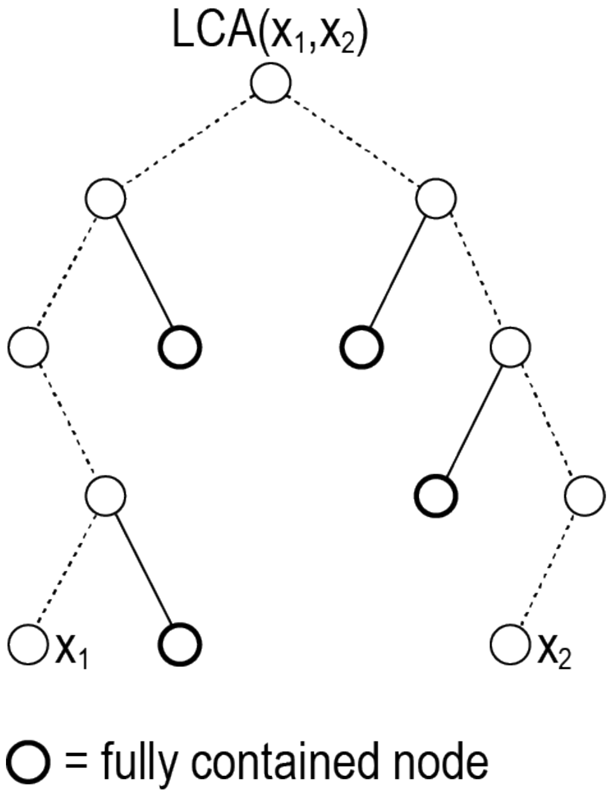

Figure 3.2: Traversing left from the LCA, each right subtree contains x-coordinates between x_1 and x_2. Traversing right from the LCA the same holds for left subtrees. Dotted line represent the path from the lowest common ancestor of x_1 and x_2 to x_1 and x_2

3.2 Original Range Search

This section describes the ORS data structure. While the theoretical work of this thesis is a simplification of this data structure, the ORS can also be viewed as an extension of the SRS. The ORS data structure will not be implemented, but serves as a theoretical background for the SRS. The main property the SRS and ORS share is that the underlying data structure is the ball-inheritance data structure and solving the range reporting heavily relies on solving the ball-inheritance problem.

Utilizing the ball-inheritance structure, Chan et al. [2] propose a theoretically better solution for orthogonal range search queries than the one of the Simple Range Search data structure:

Theorem 2.1 *for any $2 \leq B \leq \lg^{\epsilon} n$, we can can solve 2-d orthogonal range reporting in rank space with $\mathcal{O}(n \lg_B \lg n)$ space and $(1+k)\mathcal{O}(B \lg\lg n)$ query time.*

In this section some supporting data structures will be introduced. Then we will show how the ball-inheritance is used in conjunction with these data structures to find the points within a search query $q = [x_1, x_2] \times [y_1, y_2]$.

3.2.1 Preliminaries

Range minimum queries

In order to find the smallest element in a range, a succint data structure will be used. This data structure can solve the *range minimum query* problem and will be referred to as RMQ. Consider an array A with n comparable keys, this succint data structure allows finding the index of the minimum key in the subarray $A[i, j]$. Fischer [3] introduces a data structure which solves this problem in $2n + \mathcal{O}(n)$ bits of space with constant query time. The construction requires that the array is ordered, which we will see fits into our scheme.

Rank space predecessor search

In order to look up the rank space predecessor of a given coordinate, another succint data structure will be used. This data structure has a worse space complexity than the RMQ and a bigger query time. Given a sorted array $A[1..n]$ of ω-bit integers, predecessor search queries in $\mathcal{O}(\lg \omega)$ time is supported using $\mathcal{O}(n \lg \omega)$ bits of space with *oracle access* to the entries in the array. Since $\mathcal{O}(n \lg \omega)$ bits of space is not enough to store n ω-bit integers, the data structure will only store part of each n elements and defer the look-up operation to an *oracle machine*. For this purpose a Patricia trie [5] is used to store some parts of the n ω-bit integers and find which entries in the array A which has to be looked up by the oracle machine. The oracle machine is some data structure that, given an index i, can return $A[i]$. Note that $\mathcal{O}(n)$ is less than the bits needed to store A. Thus, only the index of the minimum key in $A[i, j]$ can be returned, not the actual key.

Finding the lowest common ancestor

In the Simple Range Search, the lowest common ancestor of x_1 and x_2 was found by following the path from the root of the tree to both x_1 and x_2. The last node both paths shared was then the lowest common ancestor. This way of finding the lowest common ancestor takes $\mathcal{O}(\lg n)$ time which is too big for the Original Range Search. In order to look up the lowest common ancestor of x_1 and x_2 in constant time, we are going to look at the binary representation of x_1 and x_2. First we look at $x_1 \oplus x_2$. The number of zero bits from to the start to the first one bit describes the amount of nodes the two paths have in common. We denote this number b. This will tell us to look on level $\lg n - b$. The number that the first b bits of x_1 describes is the identity of the lowest common ancestor of x_1 and x_2 on level $\lg n - b$. Modern machines have an instruction for finding the most significant set bit of a number in constant time. Thus, the lowest common ancestor of x_1 and x_2 can be found in constant time.

3.2.2 Solving range reporting

With a solution to the ball-inheritance problem, Chan et al. [2] propose the following:

Lemma 2.4 *if the ball inheritance problem can be solved with space S and query time τ, 2-d range reporting can be solved with space $\mathcal{O}(S + n)$ and query time $\mathcal{O}(\lg\lg n + (1 + k)\tau)$.*

The ball distribution scheme of this data structure is the same as the simplified range search of section 3.1.2. Having distributed the n points from the root to the leaves, additional data structures are required in order to answer the range queries. For each node in the tree that is a right child a range minimum query structure is added. The indices are the y-rank and the keys are the x-rank that the given node contains. A range maximum query structure is added to all the nodes which are left children. Each data structure uses $2n + \mathcal{O}(n)$ bits, making it $\mathcal{O}(n)$ bits per level of the tree and $\mathcal{O}(n \lg n)$ bits in all - i.e. $\mathcal{O}(n)$ words of space.

In order to support predecessor (and successor) search for the y-rank in the data structure, the rank space predecessor search data structure is added to the tree. This data structure works on an array of the y-ranks, which is already sorted. The points in rank space of $\mathcal{O}(\lg n)$ bits will use $\mathcal{O}(n \lg\lg n)$ bits per level, with $\omega = \lg n$, and $\mathcal{O}(n \lg n \lg\lg n)$ bits in all, which is $\mathcal{O}(n \lg\lg n)$ words. In order to reduce this to linear space we will only place this predecessor search structure at levels which are multiples of $\lg\lg n$. When using the predecessor search from the lowest common ancestor of $\hat{x}_1$ and $\hat{x}_2$, $LCA(\hat{x}_1, \hat{x}_2)$, we go up to the closest ancestor node which has a predecessor structure in order to perform the search there. Searching takes $\mathcal{O}(\lg\lg n)$ time plus $\mathcal{O}(1)$ queries to the ball-inheritance structure. The ball-inheritance structure is exactly the oracle access that the rank space predessecor search needs: At any given node the balls are sorted by their y-rank and given a the identity (the index) of a ball at that node, it can look up the y-coordinate. **►y-coordinate of . . .◄** Using the ball-inheritance structure we walk at most $\lg\lg n$ steps down while translating the ranks of y_1 and y_2 to the right and left child of $LCA(\hat{x}_1, \hat{x}_2)$.

The reason why this structure is necessary for the y-ranks and not the x-ranks, is because of the way the points have been distributed in the ball-inheritance tree: From left to right, the leaves have x-rank $1, 2, ..n$ so we can easily locate a given range in the x dimension, but in order to keep track of the y-dimensional range we need to follow the balls down the ball-inheritance structure. Adding this structure to each $\lg\lg n$ level saves us from going all the way from the root down to the LCA.

In order to use this data structure to report points in the range of $q = [x_1, x_2] \times [y_1, y_2]$ we follow these steps:

1. We find the rank space successor of x_1 and the rank space predecessor of x_2. We call them $\hat{x}_1$ and $\hat{x}_2$. We use these to find the lowest common ancestor of $\hat{x}_1$ and $\hat{x}_2$, $LCA(\hat{x}_1, \hat{x}_2)$. This is the lowest node in the tree whose subtree contains at least all the points between x_1 and x_2. By knowing $\hat{x}_1$ and $\hat{x}_2$, finding the lowest common ancestor is a constant time operation.

2. As in step 1 where we found the rank space of the x-coordinates, we find the rank space coordinates of the y-coordinates, $\hat{y}_1$ and $\hat{y}_2$, inside the left and right child of $LCA(\hat{x}_1, \hat{x}_2)$. We are only interested in the rank of y_1 and y_2 among the points stored in the left and right subtree of $LCA(\hat{x}_1, \hat{x}_2)$. This step is precisely what the rank space predecessor structure mentioned above supports.

3. We now descend into the right child of $LCA(\hat{x}_1, \hat{x}_2)$ and use the range minimum query structure to the find the index m (the y-rank) of the point with the smallest x-rank in the range $[\hat{y}_1, \hat{y}_2]$. The y-rank of a point at a node is exactly the identity of the ball going to the leaf storing the point. Knowing the identity of the ball we can use the ball-inheritance structure to follow the path to the leaf to find the actual x-coordinate of the point. If the x-coordinate is smaller than x_2 we return the point as a result and recurse into the ranges of $[\hat{y}_1, m-1]$ and $[m+1, \hat{y}_2]$ in order to find more points. Otherwise we terminate. When this is done we apply the same concept to the left child of $LCA(\hat{y}_1, \hat{x}_2)$ using the range maximum query to find points above x_1.

►Insert figure to conceptually show we are working our way out from the inside◄

The time complexity of step 3 depends on the use of the ball-inheritance structure. The time to traverse this structure is dependent on the improvements made in 3.1.2. An empty range will result in two queries, one query to each child of $LCA(\hat{x}_1, \hat{x}_2)$. In the worst case the amount of queries to the ball-inheritance structure will be twice the number of results reported plus one. Each time a result is found, a recursion is made to both the left and right subrange of that result. If one of the sides constantly fails to find a result, at most two queries are made for each result found. For the final result found, two ranges are recursed into which reports no results.

Conceptually, $LCA(\hat{x}_1, \hat{x}_2)$ describes a point between x_1 and x_2. Step 3 selects points that are in the range of $[y_1, y_2]$ moving outwards from the point of $LCA(\hat{x}_1, \hat{x}_2)$, always picking the point closest to $LCA(\hat{x}_1, \hat{x}_2)$ in its decreasing y-range. **►rephrase◄**

Going back to Lemma 2.4, we see that the time complexity fits: $\mathcal{O}(\lg \lg n)$ time is used for the predecessor search and $\mathcal{O}((1+k)\tau)$ time is used for walking from the children of the LCA to the leaves solving the ball inheritance problem for the k results.

3.3 Summary

The SRS and ORS data structures both uses $\mathcal{O}(n)$ words of space. Both relies heavily on the ball-inheritance data structure in order to store and retrieve the actual coordinates of points. Both data structures supports retrieval of points through the ball-inheritance data structure in $\mathcal{O}(\lg^{\epsilon} n)$ time. The main difference between the SRS data structure and the ORS data structure is the rest of the query time: $\mathcal{O}(\lg n)$ for the SRS and $\mathcal{O}(\lg \lg n)$ for the ORS. The ORS data structure relies on more auxillary data structures of greater complexity than the SRS data structure. Thus, the theoretical running time of a range query to ORS is smaller than a range query to the SRS. In practise, it is safe to assume a range query to the SRS is faster than a range query to the ORS. With much more code to be executed and using more advanced auxillary data structures, the running time constant hidden in $\mathcal{O}(\lg \lg n)$ can be quite large. Also, the factor between $\lg n$ and $\lg \lg n$ is not that big. Given a very large dataset of 2^{64} elements as input, $\lg 2^{64} = 64$ and $\lg \lg 2^{64} = 6$, which has a factor ~ 10 difference.

The SRS and ORS data structures supports search queries in a manner similar to the range tree. A balanced binary search tree is used to locate the subtrees which is fully contained in $[x_1, x_2]$. From here the range tree uses balanced binary search trees or fractional cascading to locate which of those points are in $[y_1, y_2]$, while the SRS and ORS data structures uses the ball-inheritance structure to decode the y-ranks to actual points.

Both the SRS data structure and the kd-tree uses $\mathcal{O}(n)$ words of space. This is an attractive property when working on the RAM. Like all the main data structures mentioned in this thesis, the kd-tree and SRS data structure are output-sensitive. A range query to the kd-tree has a running time of $\mathcal{O}(\sqrt{n}+k)$ and a range query to the SRS data structure has a running time of $\mathcal{O}(\lg n+(1+k)\cdot \lg^{\epsilon} n)$. When n grows, $\mathcal{O}(\sqrt{n})$ grows at a faster rate than $\mathcal{O}(\lg n)$. For each point reported as a result from a range query to the SRS data structure there is a cost of a factor $\mathcal{O}(lg^{\epsilon}n)$ while each point reported as a result from a range query to the kd-tree has a cost of a factor $\mathcal{O}(1)$. The $\mathcal{O}(\sqrt{n})$ running time of the range query to the kd-tree is due to a pessimistic idea that a range query will overlap, but not fully include, a lot of regions in the kd-tree. So dependent on the shape of the range query to the kd-tree, the running time can vary a lot. The $\mathcal{O}(\lg n)$ part of the range query to the SRS data structure is due to the initial binary search and to follow the path from the root of the tree to x_1 and x_2. Thus, the running time of a range query to the SRS data structure will behave more stable (fluctuate less) than the running time of a range query to the kd-tree.

▶Kan man sige noget en sammenligning af de to worst-cases?

At den ene er bedre end den anden og det sætter en bedre upper bound?◀

Another aspect of range querying is testing for emptiness. When testing for emptiness, a range query can stop the moment a single result is found since it is a test with the binary outcome of "yes" or "no". The SRS data structure will be able to determine emptiness of a range query with no ball-inheritance queries. Hence, an emptiness query to the SRS data structure can be checked in $\mathcal{O}(\lg n)$ time. The initial binary search to find the rank space query might indicate that there are no points in $[x_1, x_2]$ or $[y_1, y_2]$. If the initial binary search does not indicate an empty range, the range query will continue normally. The moment a query to the ball-inheritance structure is made, the range is not empty and the emptiness test can report back without doing the actual ball-inheritance query. An emptiness query to the kd-tree can checked in $\mathcal{O}(\sqrt{n})$ time. The same argument as before applies: The query might overlap with a lot of regions which it does not fully contain, and within those regions none of the points are within the range. On the other hand, when a fully contained region is found the emptiness test can report back with a result. Given a range query $[x_1, x_2] \times [y_1, y_2]$, the SRS data structure will have a more stable running time for its emptiness test than the kd-tree. The running time of a query made to the kd-tree will fluctuate a lot depending on the shape of the query.

▶Ting du skal huske at have med i dette kapitel: Decode y-rank i stedet for at slå op i søgetræ. Range-tree imod ball-inheritance◀ ▶Balls per level vs balls per node - hvordan skal det introduceres (og hvor). Rank af bolde i node (per node)◀

Part II

Practical

Chapter 4

Implementation

4.1 Language

C++11 was chosen to implement the *SRS* data structure and the kd-tree. C++11 is a recent release of C++. C++11 combines the advantages of a modern programming language with the stability and support of a mature language which have been around for more than three decades. C++11 was chosen for several reason: It does not have garbage collection, it is a high level language with support for low level operation and contains libraries for everything needed in this project. The chrono header gives access to a high resolution clock to meassure time. The vector class of the standard template library is very straight-forward container to work with and the underlying structure is a continuous block of memory making it ideal for cache purposes. The algorithm header gives access to convenience functions for generating and sorting data. The random header gives access to random numbers through a Mersenne twister engine with uniform integer distribution. The random numbers are used to generate the input data for the two data structures, thus giving a different data set every time.

4.2 Design choices

The theory describes the ball inheritance data structure as a binary tree with internal nodes and leaves. Each node has a bit vector representing the ball inheritance. In practise all the bit vectors at each level have been concatenated together to one, resulting in $\lg n$ bit vectors in all. Instead of being an actual entity, a node is just defined as which level it is from, where on that levels list its bit vector starts and how many balls its bit vector holds. Thus, a vector of bit vectors represents the ball inheritance tree. Since each ball only uses a single bit per level there would have been a lot of space wasted when nodes only held two or four balls in their bit vectors. By having a single unified bit vector per level we can pick one data type to work with independent of how many balls needs to be stored per node. The data type chosen is an unsigned fixed width integer type of 32 bits.

We are going to describe how to support constant-time succint rank queries for a unified bit vector. First a checkpoint is added to every 32nd entry storing the amount of 1s seen in the bit vector so far. Since the bit vectors consist of 0s and 1 we only need count the amount of 1, since the amount of 0s can be thought of as *not*-1*s*. A table is computed which given a 16 bit unsigned integer is able to answer how many 1 are in the binary representation of the integer. The table is a flat array and supports look-up in constant time.

Given a position i in the bit vector, the closest checkpoint previous to i is found. The data type storing the bits in the bit vector is a 32 bit unsigned integer. We can then only retrieve 32 bits from the bit vector at a time**▶ord?◀**. We use a *binary and* to mask away the bits after i and divide the 32 integer into two 16 bits integers. Using the precomputed table from before, we look up how many 1s the binary represention of the two 16 bit integer contain. **▶EXAMPLE◀**. The sum of the major checkpoint and the two table look-ups is the amount of times a 1 occurs before position i in the bit vector.

However, we are seldomly interested in knowing how many 1s there are in a bit vector between position 0 and i. We want to know how many 1s there are between j and i, where j is the start position of a node and i is an entry in that node. We know that with the ball inheritance structure that each node gives each of its children half of its balls. Thus, half of the entries in the bit vector of a node are 1s. And thus, if j is the start position of a node in the unified bit vector, there will be $\frac{j}{2}$ entries in the bit vector between 0 and j which are 1s. Since one of the ingredients for a virtual node is the start position in the unified bit vector, we already know j when looking up the rank of i. Using the unified bit vectors instead of an actual tree then poses no problems.

Skriv om hvordan det virker med enkelte hop. Vi har en look-up til 16 bits. Vi har en sum for hvert 32 bits. At 0 bare er komplementar af 1 når vi tæller.

4.3 Ting der skal sættes ind hist og her

In section ref-to-kd-tree the $\mathcal{O}(\sqrt{n}+k)$ running time of kd-tree search was explained to be due to a pessimistic idea that a search query would make a line through the entire region of the root. Thus, this is the worst case for a search to the kd-tree. In order to test the kd-tree and the SRS in this situation, we introduce the term *slice*. A slice is a search query which extends through the entirety of the point region **▶andet ord◀** in one dimension while being narrow on the other dimension. A slice which extends through the entirety of the y-dimension is called a *vertical slice*. A slice which extends through the entirety of the x-dimensio is called a *horizontal slice*. When describing the size of a slice, we will only specify the size of its narrow dimension, since the other is already known.

In subsection 3.1.2 we saw how we could speed up the ball-inheritance by allowing bigger jumps by using a bigger alphabet. As introduced in subsection 3.1.1 a bigger alphabet comes with a cost. Looking at the implemention of the extended alphabet and the binary packing we can set an upper bound on the storage cost of the multi-level jumps. Per level, we can define the space cost of big jumps, in bits, as:

$$\begin{aligned} checkpoint_{major} &= \frac{n}{\Sigma \lg n} \cdot \Sigma \lg \frac{n}{\Sigma} \\ checkpoint_{entry} &= n \lg(\Sigma \lg n) \\ entry &= n \lg \Sigma \end{aligned}$$

When a jump reaches a leaf, the rank of the ball is not needed and thus, we can do without the two checkpoints. This way we can save a lot of space. The space has already been budgetted for, and thus we can use it to extend the size of the jumps or even add smaller jumps at other levels.**►Tilføj ulighed - Vis at et hop kan blive til to på samme plads eller mindre◄**

We have already looked at the space complexity of the both the kd-tree and the SRS, but if we look a little closer we can argue about the constants hidden away in $\mathcal{O}(n)$. **►noget med 32 vs 64 bits - plus at $n \lg n$ i ball inheritance ikke er n words. Det er nemlig pakket og derfor bare boundet af n◄**.

►Skal det her overhovedet med?◄ For smaller alphabets, another scheme is used. But $\sqrt{\lg n}$ is already pretty small, so instead of implementing a whole other scheme just in order to skip 1 or 2 levels, we just travel normally until we find a level containing jumps with a $\Sigma \geq \sqrt{\lg n}$ **►Eller $\Sigma \geq \lg^{\epsilon} n$◄**. This should not have a big impact. **►rephrase◄**

Chapter 5

Analysis

The purpose of this chapter is to compare the SRS data structure to the kd-tree. We are going to perform a variety of tests on the SRS data structure and the kd-tree in order to determine the run-time properties of both, mainly looking at when the SRS data structure performs better than the kd-tree.

In order to compare these two data structure random data will be generated and given as input to both, such that both operate on the exact same data. When running a specific test, different data sets are generated during such that one tests is not only performed one a single data set. When the size of a slice has been determined, the search will be performed different places in the data structure such that not only best-case or worst-case scenarios occur. Finally the average of all the searches are returned as the result of the test.

5.1 Vertical and horizontal slices

Before introducing the graphs, we are going to look at the theoretical run-time of a search query to the kd-tree and the SRS data structure. A search query to the kd-tree has a run-time of $\mathcal{O}(\sqrt{n} + k)$ and a search query to the SRS data structure has a run-time of $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$. When increasing the size of a slice, we expect the run-time of the two search queries to be roughly equal at $k \approx \frac{sqrtn}{lg^{\epsilon}n}$ stemming from $\sqrt{n} + k = \lg n + k \cdot \lg^{\epsilon} n \Leftrightarrow k = \frac{\sqrt{n} - \lg n}{\lg^{\epsilon} n - 1}$. The $\mathcal{O}(\lg^{\epsilon} n)$ describes the amount of jumps needed to perform in order to find the identity of a leaf given the identity of a ball in the ball inheritance structure. Thus, the size of $\mathcal{O}(\lg^{\epsilon} n)$ on how B is chosen in **►ref◄** and which other meassures have been taken in order to lessen the amount of jumps. **►Skriv om at vi har målt dem også, og at de er sat ind på en graf - det er de værste hop vi kigger på? eller er det gennemsnittet? Hvad med grafen der viser $\frac{Sk\_ringspunkt}{\frac{\sqrt{n}}{lg^{\epsilon}n}}$ - skal det være worst $\lg^{\epsilon} n$ eller gennemsnittet?◄**

Recall that the SRS data structure treats the two dimensions very differently. Given a rank space search query $\hat{q} = [\hat{x_1}, \hat{x_2}] \times [\hat{y_1}, \hat{y_2}]$, the search algorithm will find the least common ancestor of $\hat{x_1}$ and $\hat{x_2}$. From there, it will find the path to both $\hat{x_1}$ and $\hat{x_2}$ and all leaves between them will be in the range $[x_1, x_2]$. The search algorithm now has to determine which of these leaves contain a point with y-coordinate in $[y_1, y_2]$. This is done by using the ball inheritance structure from each of the fully contained nodes which were found on the path from the least common ancestor to $\hat{x_1}$ and $\hat{x_2}$.

This means that a horizontal slice and a vertical slice will be treated differently by the SRS data structure. The search query of a horizontal slice includes all x-coordinates of the search space, and thus the least common ancestor of $[\hat{x_1}, \hat{x_2}]$ will be the root of the tree. The path from the least common ancestor to $\hat{x_1}$ will only go left which means each level will give a fully contained node. The path from the least common ancestor to $\hat{x_2}$ will only go right, also yielding a fully contained node per level. The tests are to measure how big a slice can become before the kd-tree performs just as well. This means that a slice will start out being very small, $k = 5$. Thus, many of the fully contained nodes will have no ball inheritance work. As the slice grows, eventually each node will have one or more balls to follow. The amount of ball inheritance each node is responsible for varies a lot, and therefore the ball inheritance will become somewhat sporadic.

On the other hand we have the vertical slice. The vertical slice includes all y-coordinates of the search space, and thus the location least common ancestor of $[\hat{x_1}, \hat{x_2}]$ varies a lot dependent on the search query. But the nodes which are marked as fully contained on the path from the least common ancestor to $\hat{x_1}$ and $\hat{x_2}$ will all only contain leaves with points with y-coordinates in $[y_1, y_2]$ which means we have to do ball inheritance on all the balls belonging to fully contained nodes. Thus, the ball inheritance in vertical slices are much more batched together. Comparing a vertical and horizontal slice of the same size, the vertical slice will have good chances of being faster than the horizontal slice. With a lower LCA, the ball inheritance in the vertical slice will have to jump from a lower level than the ball inheritance in the horizontal and the balls are more batched together in the vertical.

The first thing to be tested is how big the vertical and horizontal slices can become before the SRS data structure performs worse than the kd-tree. We are going to look at the vertical slices first. Below are some graphs showing the running time of a search query to both the SRS and the kd-tree dependent on the size of the slice. Recall from section **►ref◄** that the parameter B could be set such that we could fix the size of the alphabet (and the jumps) in the ball inheritance tree. The graphs below shows results from a SRS data structure configured with $B = 2$.

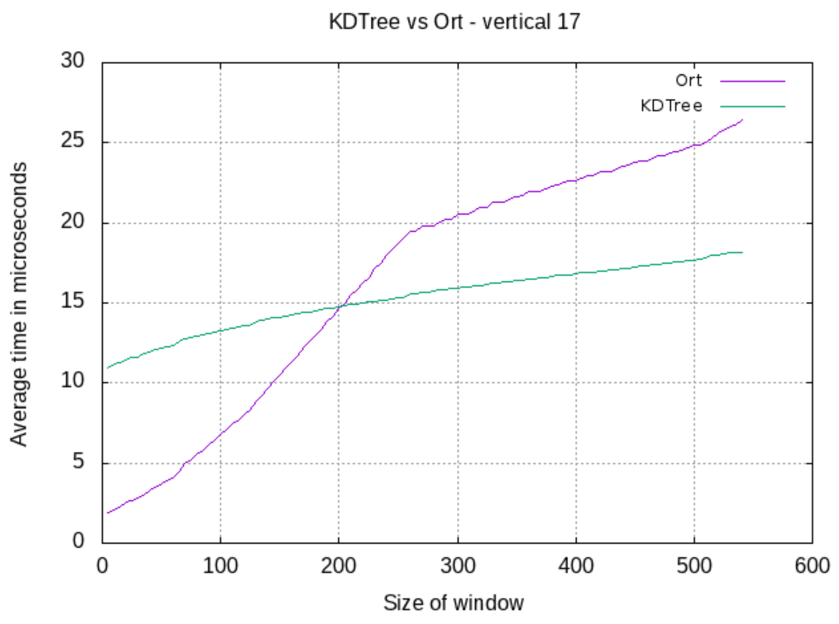

Figure 5.1: vertical slice on SRS and kd-tree - data set size of $n = 2^{17}$

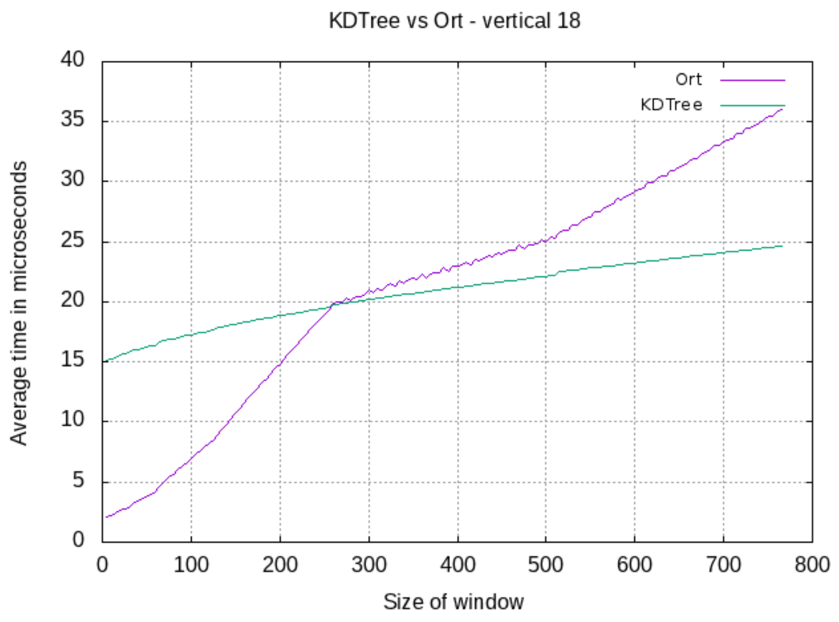

Figure 5.2: vertical slice on SRS and kd-tree - data set size of $n = 2^{18}$

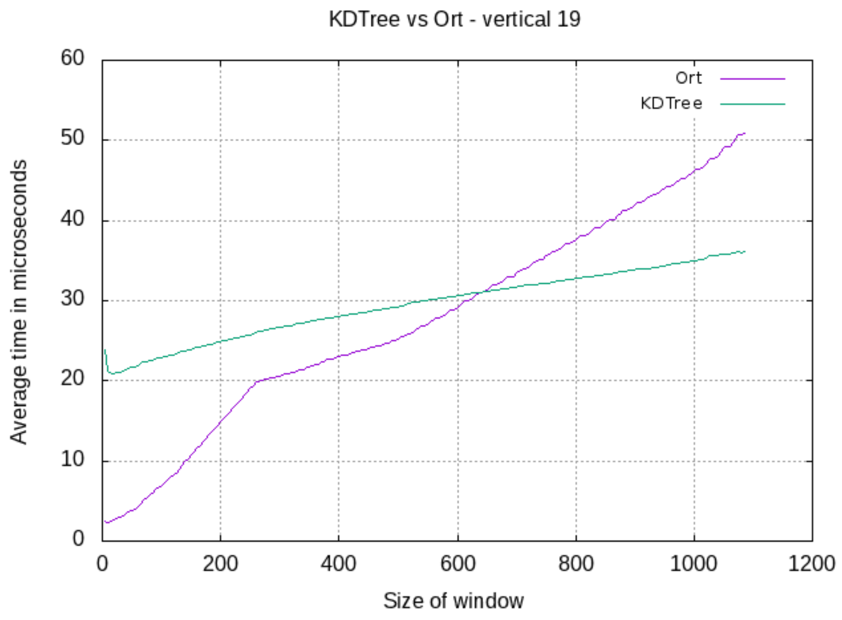

Figure 5.3: vertical slice on SRS and kd-tree - data set size of $n = 2^{19}$

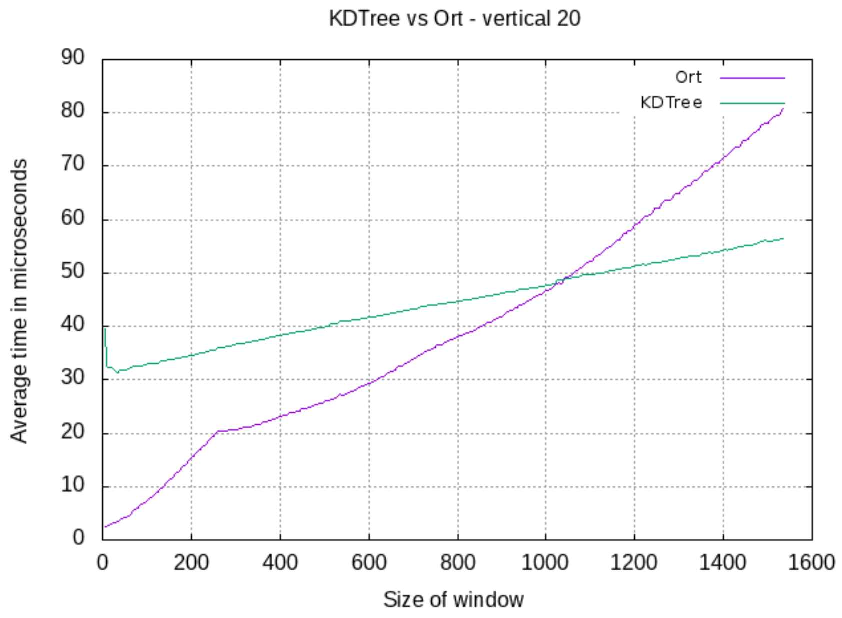

Figure 5.4: vertical slice on SRS and kd-tree - data set size of $n = 2^{20}$

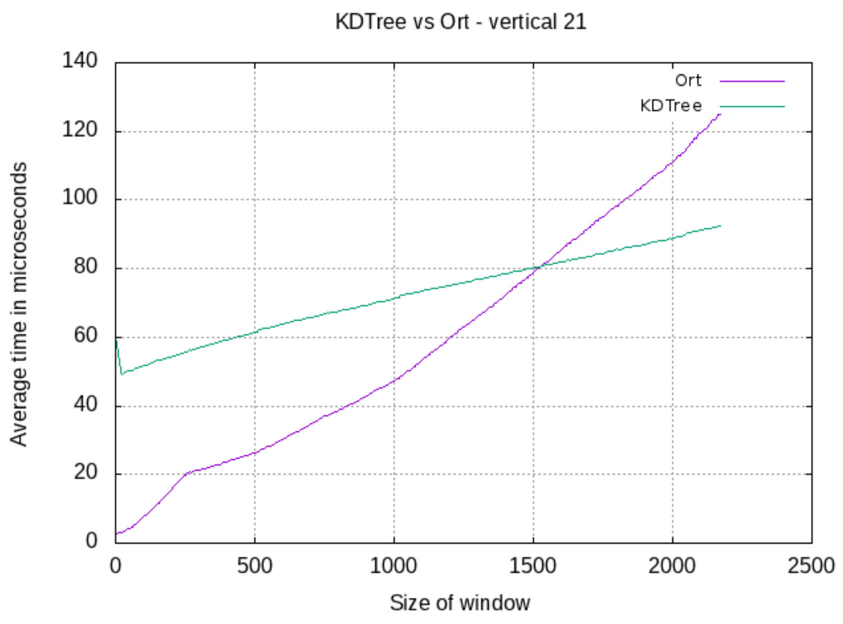

Figure 5.5: vertical slice on SRS and kd-tree - data set size of $n = 2^{21}$

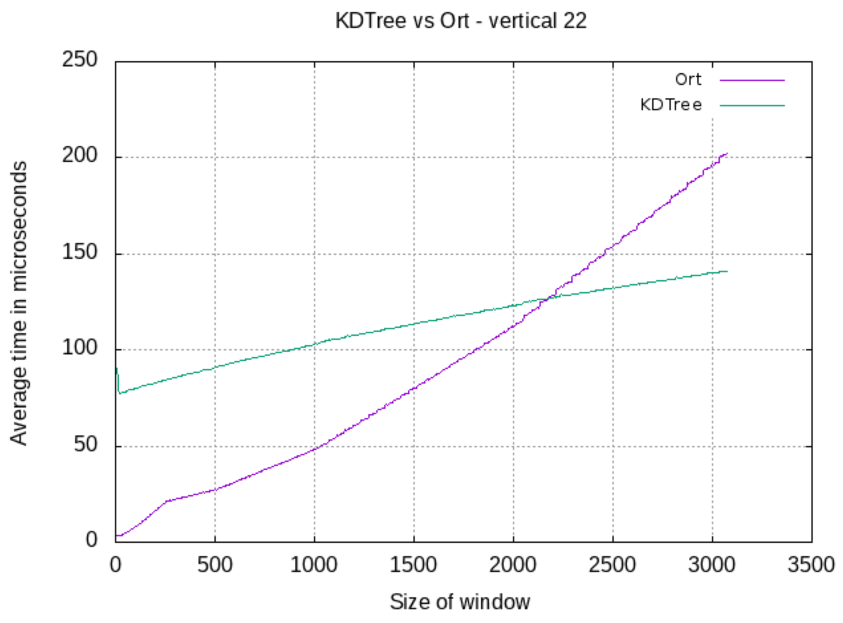

Figure 5.6: vertical slice on SRS and kd-tree - data set size of $n = 2^{22}$

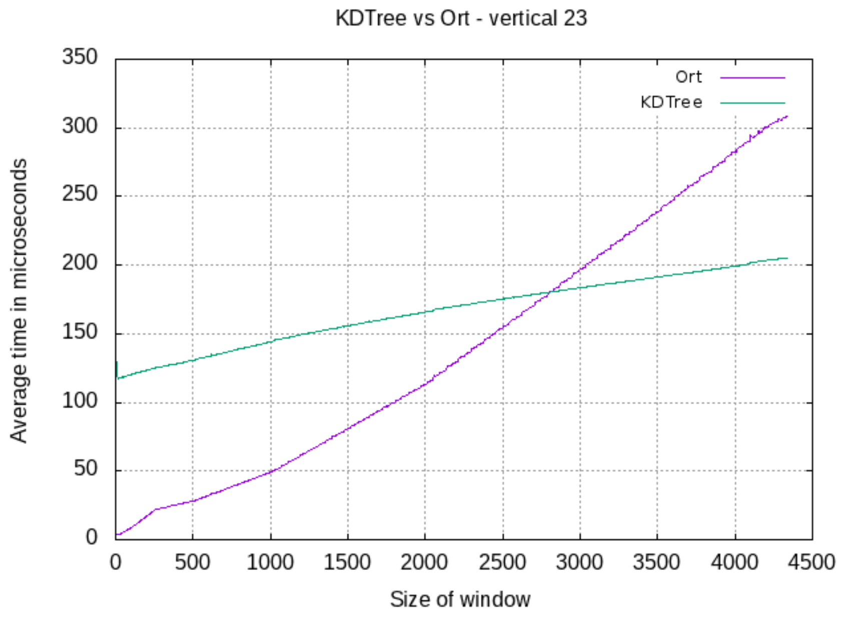

Figure 5.7: vertical slice on SRS and kd-tree - data set size of $n = 2^{23}$

Across these graphs, there is a very noticable change in slope at around $k = 256$ for the running time of a search query to the SRS data structure. Since $B = 2$, we have a big jump at level $2^3 = 8$ which allows the ball inheritance to jump from level 8 to a leaf in one jump. This means that the average amount of jumps per result will decrease and thus the running time will not increase as fast as before. At around $k = 512$ the running-time resumes its normal behaviour. On average, the ball inheritance structure does not use the big jump at level 8 directly anymore, resulting in a couple of steps before reaching level 8. Since the graph shows the average run-time of different configurations of the same slice on different data sets, not all of the searches will hit level 8 from $k > 256$, but the main tedency is to. **►forklar tidligere - at vi snakker om main tendency, ikke hvad den er garanteret at gøre◄**. This tendency is described at figure 5.8 and figure 5.9. We see how the graph has a local maximum at around $k = 256$ and then the average amount of jumps per result decreases until $k = 512$ where it starts increasing at steady level again. Figure 5.10 and figure 5.11 describes the highest level of a fully contained node. We see between $k = 256$ and $k = 512$ that the maximum is level 8 and the minimum is level 7 and that the average level increases meaning more and more fully contained node starts using level 8. Since we $B = 2$, there is a 2-jump every 2 levels, a 4-jump every 4 levels, a 8-jump every 8 levels and a 16-jump every 16 levels. This means that level 7 the ball inheritance structure needs 3 jumps to reach a leaf. This is why such a noticable local maximum exits on figure 5.8 and figure 5.9. The jumps per results eases off because from level 8 there is 1 jump, from level 9 there are 2 jumps and from level 10 there are 2 jumps. Levels $11, 13, 14$ have 3 jumps, and at level 15 another local maximum is going to be found with 4 jumps, just before level 16 with 1 jump to the leaves. However, $k = 2^{16} = 65,536$ is not likely to be a place where the SRS data structure performs better than the kd-tree unless we have a enormous data set **►regn på det◄**. **►Sæt grafer ind for større n◄**

Looking at figure 5.10 we see that when we reach $k = 256$ the minimum highest level rises to 7 and the maximum highest level rises to 8. In order to write the sum of $256 < k < 512$ we will need to at least level 7 because we need the two fully included nodes from level 7 giving us $2 * 2^7 = 2^8 = 256$ nodes. If the least common ancestor is found at level 9 the first fully contained node can be found at level 7. If all the levels from level 7 to level 1 only have fully contained nodes, we get $k = 2 * 2^1 + 2 * 2^2 + \cdots + 2 * 2^7 = 508$. This is not enough to write 512, and thus somewhere between $k = 256$ and $k = 512$, the search algorithm will begin using level 10 as the least common ancestor. This is also very dependent on how the slice covers the subtrees of the least common ancestor. If the slice hits the least common ancestor right in the middle, such that the left subtree and the right subtree of the least common ancestor is of the exact same size, the level of the highest fully included node will be lower. If the slice hits the least common ancestor such there is an imbalance in the size between the left subtree and right subtree of the least common ancestor, the level of the highest fully included node will be higher, because now one subtree has to account for a bigger portion of the slice and will have to use bigger pieces. And when a subtree contains $2^8 = 256$ leaves it will have access to the jump at level 8. The maximum level of figure 5.10 is explained by the fact that it is not possible to write the sum of $k < 512$ using 512 as a summand. Recall that a vertical slice conceptually only searches for the x-coordinates of the points. All the y-coordinates are already known to be included, Thus, when a node is fully included, we get all of points in its subtree. **▶Omformuler og sæt sammen med afsnittet fra før lige ovenover da du skriver nogenlunde det samme - bare i to forskellige tankegange◀▶Lav en lignede figur som figur 3.2 til at forklare konceptet◀**

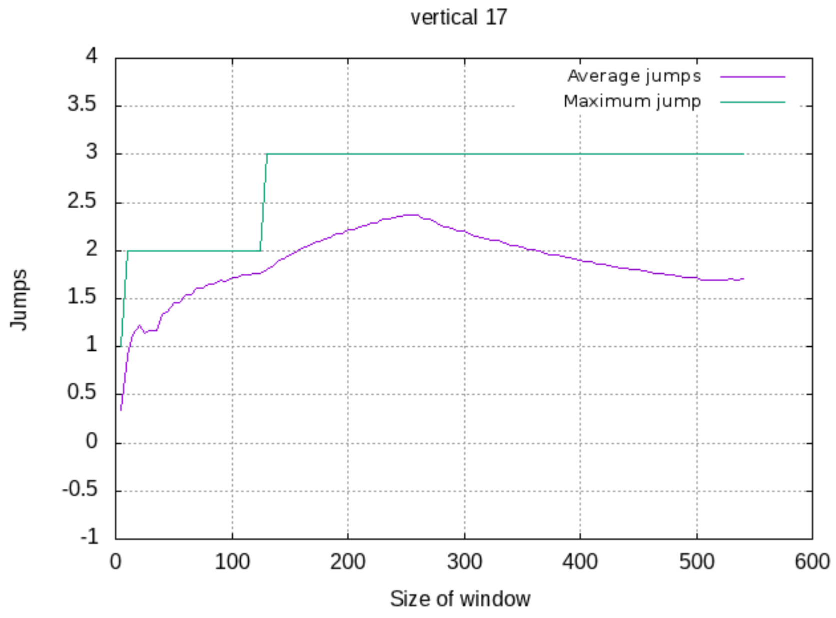

Figure 5.8: Size of jumps - data set size of $n = 2^{17}$. 'Average jumps' is the average of all the jumps performed normalized by the size of the slice

To summarize the graphs for vertical slices, figure 5.12 shows the size of the slice at the point of intersection between the running-time of a search to the SRS and the kd-tree for each n tested. Recall the theory described above where it was described how the intersection point should theoretically be $k_{theoretical} = \frac{\sqrt{n}-\lg n}{\lg^{\epsilon} n-1}$. Figure 5.26 and figure **??** show $\frac{k_{actual}}{k_{theoretical}}$. One graph uses $\lg^{\epsilon} n$ as the average jump per result while the other uses $\lg^{\epsilon} n$ as the worst case jump. If the graph is below 1 it means that the SRS data structure performed worse than theoretically expected and if it above 1 it means that the SRS data structure performed better than theoretically expected. The increase in the slope of the graph on figure 5.12 can be described by the decrease in the average jumps per results as seen on figure 5.13. This big drop in figure 5.13 can be explained by looking at figure 5.2 and figure 5.3. The intersection between the running time of the SRS and the kd-tree on figure 5.2 is just prior to the running time of the SRS changing its slope drastically. At figure 5.3 we see the of the running time of the SRS has changed prior to its intersection with the running time of the kd-tree. With access to the jump from level 8, the average amount of jumps

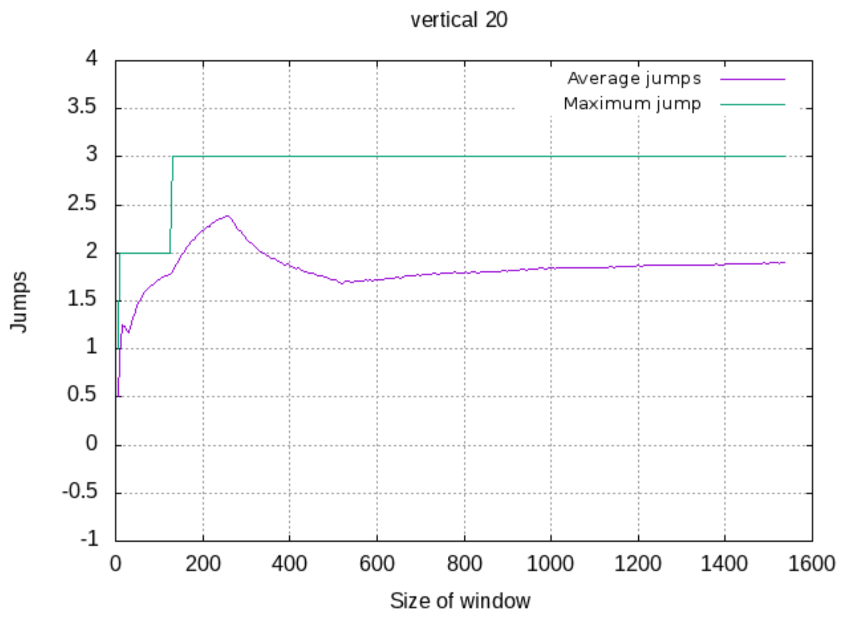

Figure 5.9: Size of jumps - data set size of $n = 2^{20}$. 'Average jumps' is the average of all the jumps performed normalized by the size of the slice

per result has been lowered. This can also be noted from figure 5.12 seeing that at $\lg n = 18$ the point of intersection is around $k = 250$.

We are now going to look at the graphs for the horizontal slices. Recall that when a search query is a horizontal slice, the least common ancestor will be the root of the tree. In $[\hat{x_1}, \hat{x_2}]$, $x \hat{-} 1$ will be the leftmost leaf and $\hat{x_2}$ will be the rightmost leaf. From the root to $\hat{x_1}$ and $\hat{x_2}$ there will many fully included nodes, 2 per level to be exact, which means there will be many different nodes performing small amount of balls inheritance look-ups. Since the points are ordered by their y-coordinate in the root before distribution, and the fact that they keep this order while being distributed means that when increasing the range from $[y_1, y_2]$ to $[y_1, y_2 + 5]$ it is more likely that these 5 new points will be found from a higher level where the range $[y_1, y_2]$ already had jumps from. It is much likely that a point is stored in a leaf which belongs to a fully included node from level 8 with 256 leaves than belonging to a node from level 2 with 4 leaves. But since the points are distributed to leaves according to their x-coordinates, it is hard to know the distribution of y-coordinates. When a search query is a horizontal slice, the position of $\hat{y_1}$ and $\hat{y_2}$ will be found on the bit vector of the root node. These position will be updated using the succint rank query while traveling from the root node to the leftmost leaf and the rightmost leaf in the tree.

▶Hvorfor knækker grafen mellem 512 **og** 1024**? Hvorfor er kd-træet mærkeligt dårligt inden** $k = 10$**? HORI◀**

To summarize these graphs for horizontal slices, figure 5.23 shows the size of the slice at the point of intersection between the running-time of a search to the SRS and kd-tree for each n tested.

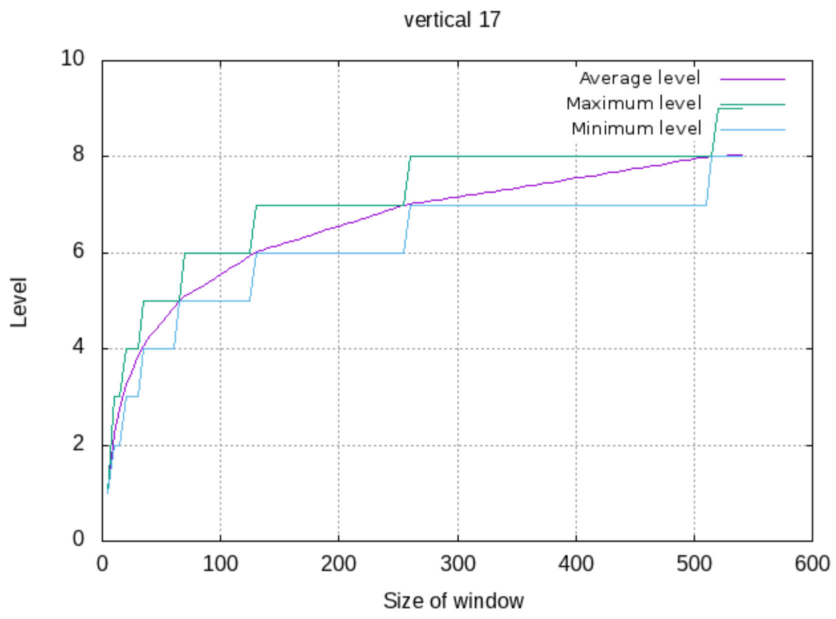

Figure 5.10: Level of first fully contained node - data set size of $n = 2^{17}$.

5.2 Comparision of SRS and kd-tree with a constantly small k

Grafer kommer

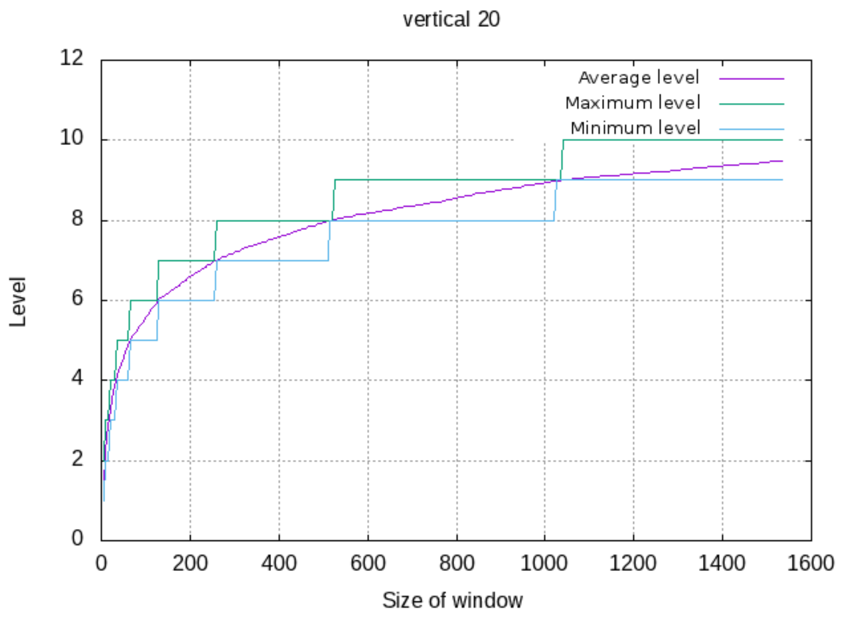

Figure 5.11: Level of first fully contained node - data set size of $n = 2^{20}$.

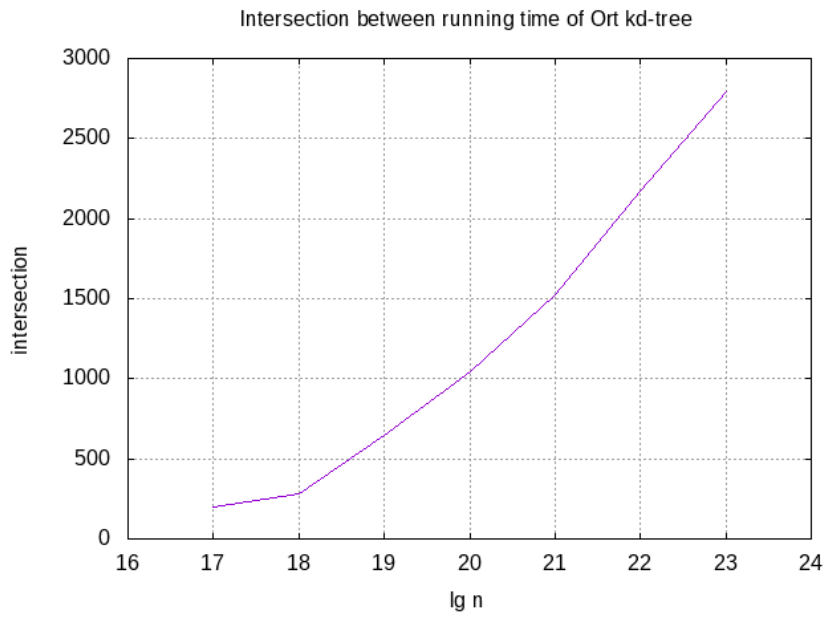

Figure 5.12: Vertical slices, intersection between running time of SRS and kd-tree

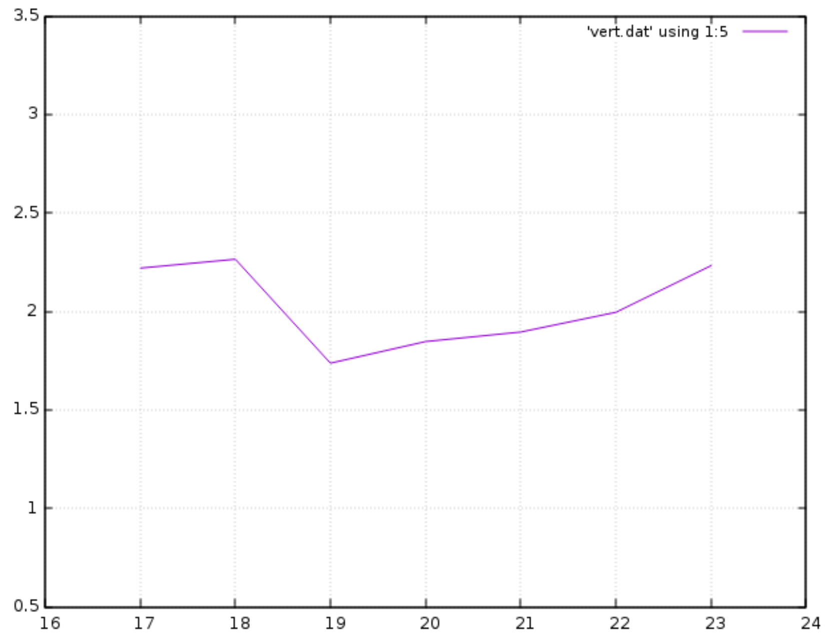

Figure 5.13: Vertical slices, average jumps per result at the point of intersection between the running time of SRS and the kd-tree

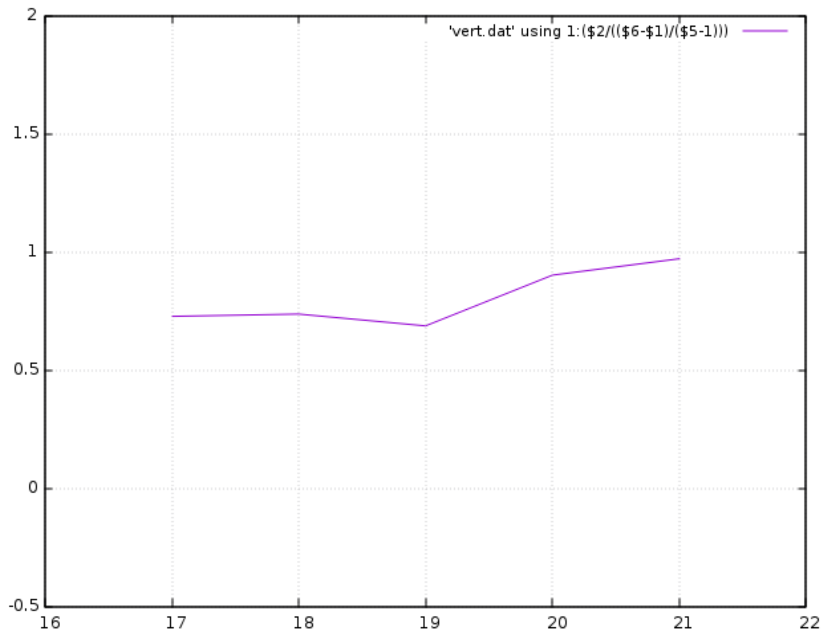

Figure 5.14: Vertical slices, intersection between running time of SRS and kd-tree normalized by $k_{theoretical}$ with $\lg^{\epsilon} n$ being the average amount of jumps per result

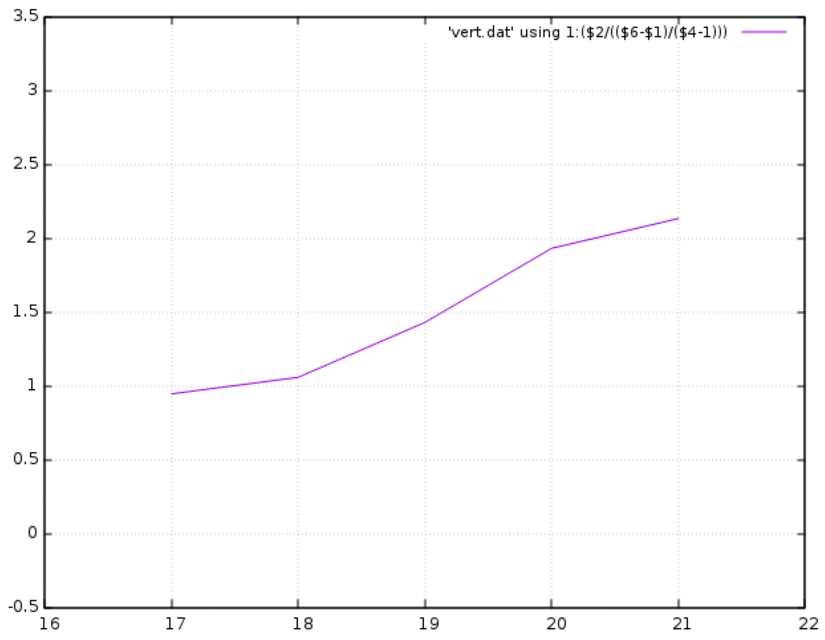

Figure 5.15: Vertical slices, intersection between running time of SRS and kd-tree normalized by $k_{theoretical}$ with $\lg^{\epsilon} n$ being the worst amount of jumps per result

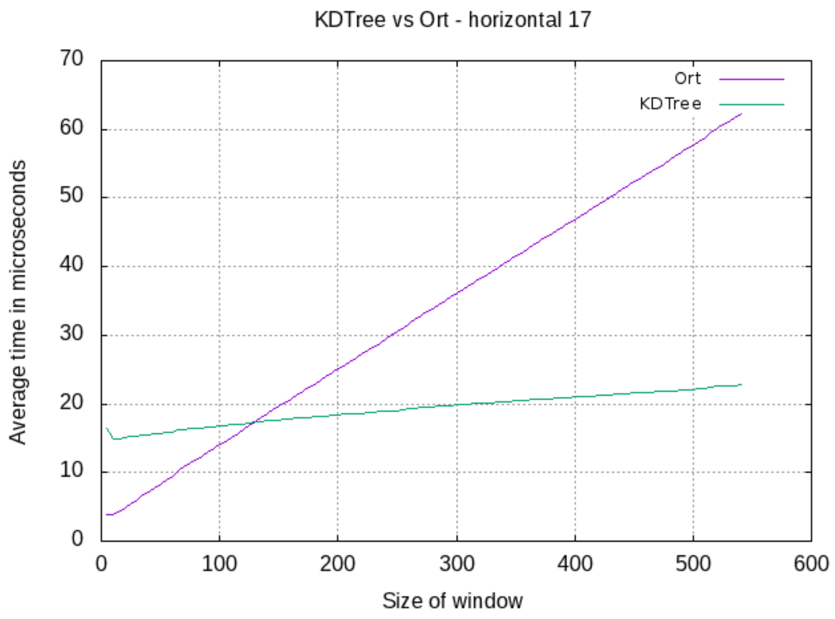

Figure 5.16: Horizontal slice on SRS and kd-tree - data set size of $n = 2^{17}$

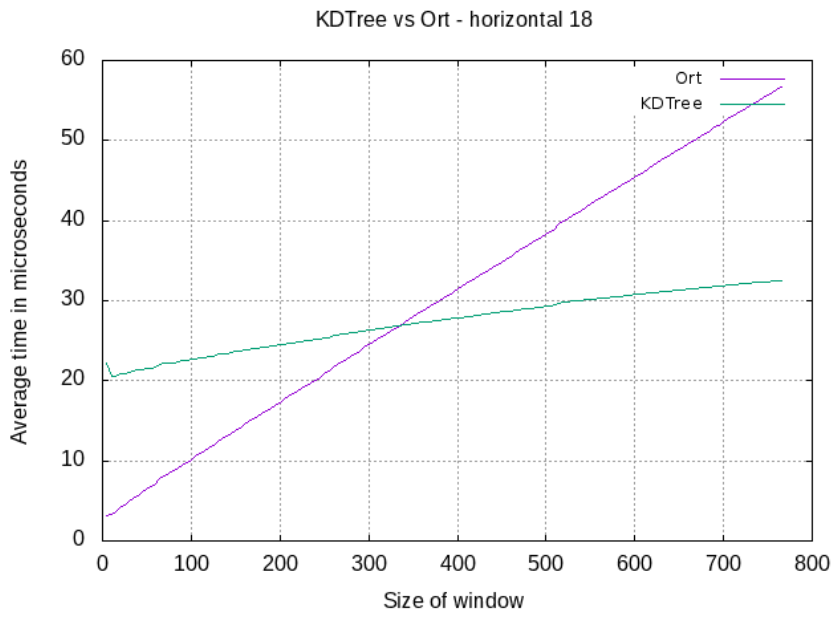

Figure 5.17: Horizontal slice on SRS and kd-tree - data set size of $n = 2^{18}$

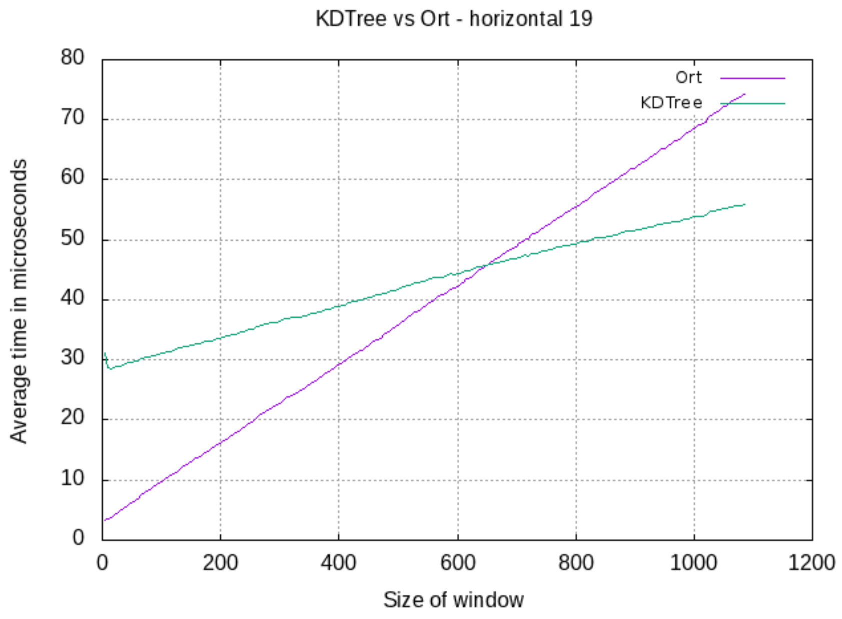

Figure 5.18: Horizontal slice on SRS and kd-tree - data set size of $n = 2^{19}$

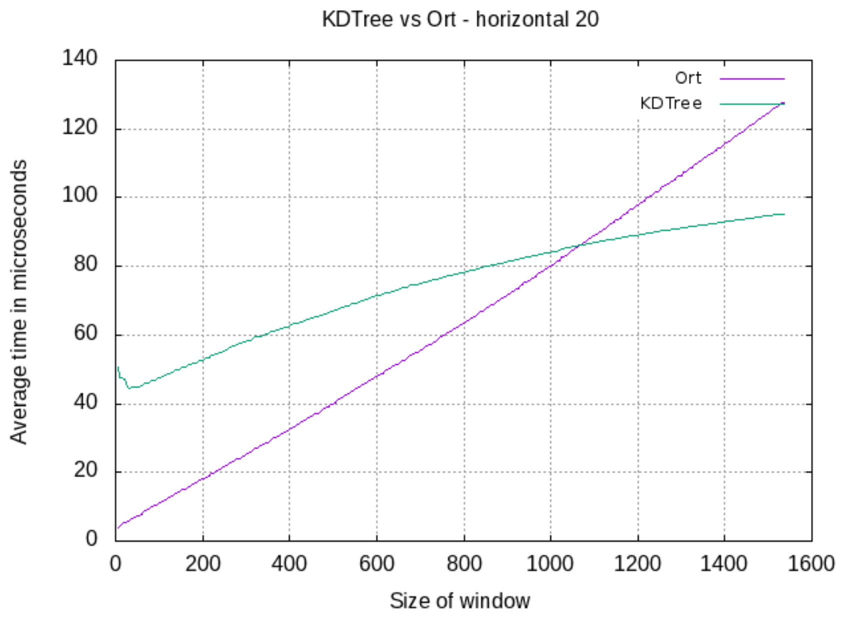

Figure 5.19: Horizontal slice on SRS and kd-tree - data set size of $n = 2^{20}$

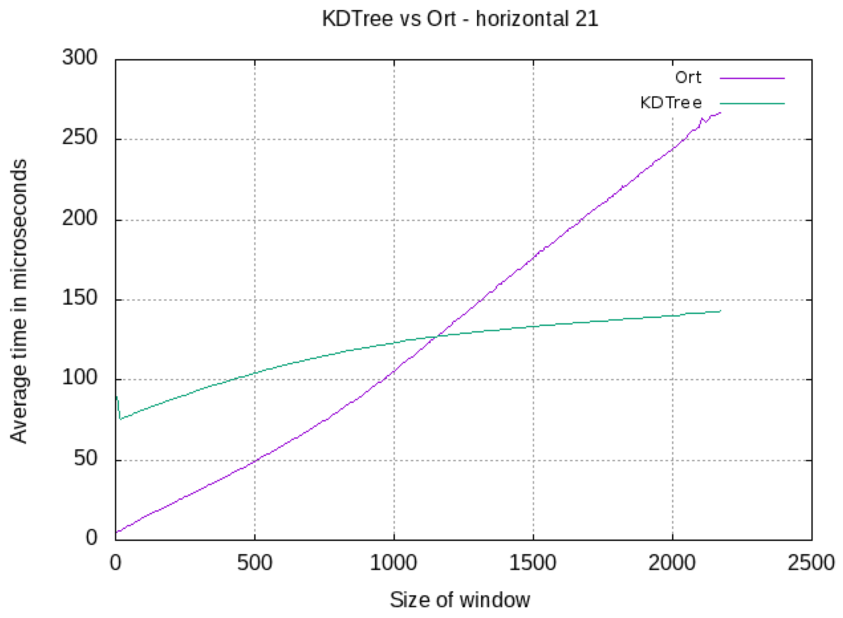

Figure 5.20: Horizontal slice on SRS and kd-tree - data set size of $n = 2^{21}$

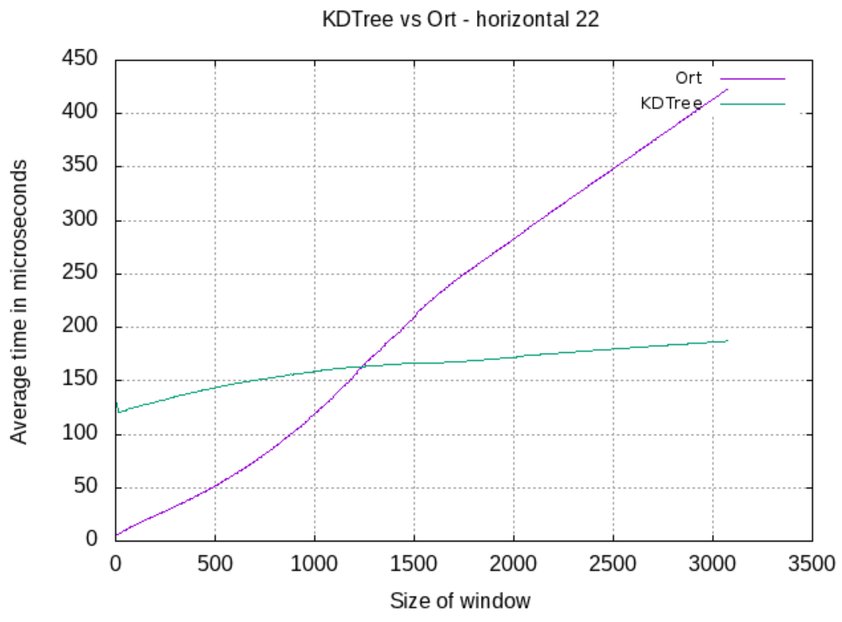

Figure 5.21: Horizontal slice on SRS and kd-tree - data set size of $n = 2^{22}$

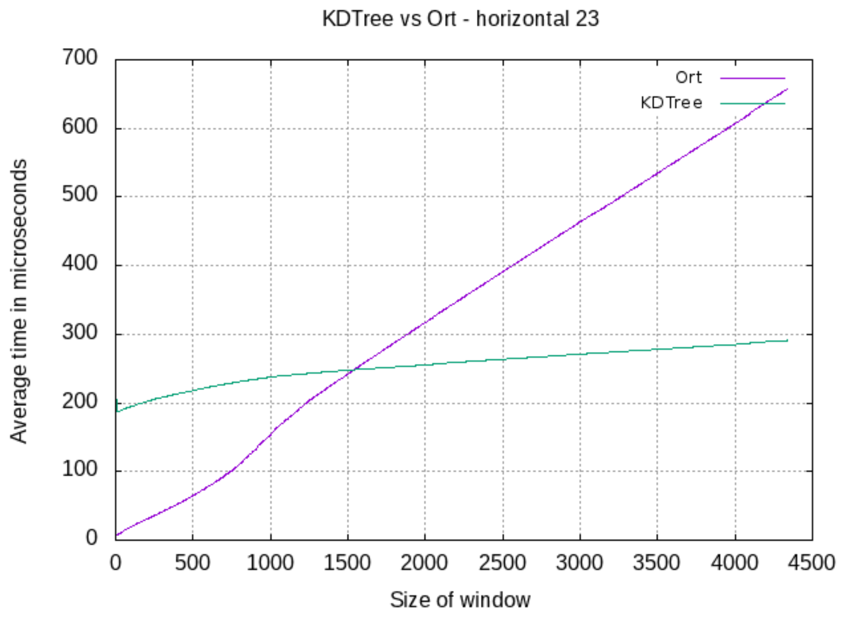

Figure 5.22: Horizontal slice on SRS and kd-tree - data set size of $n = 2^{23}$

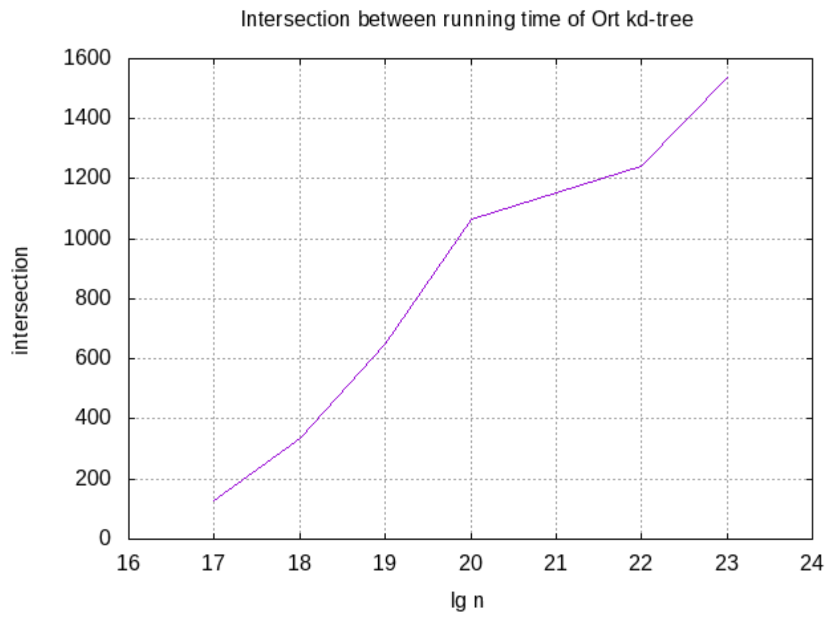

Figure 5.23: Horizontal slices, intersection between running time of SRS and kd-tree

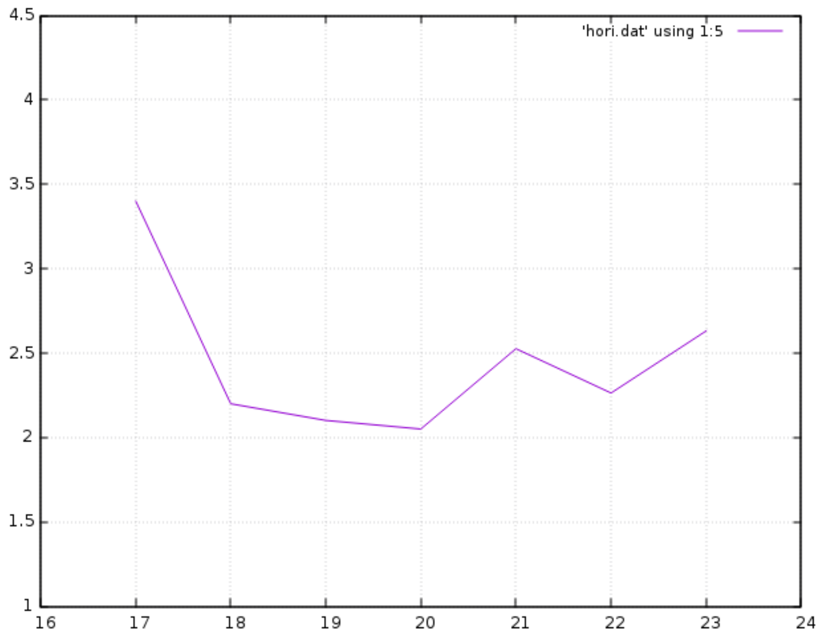

Figure 5.24: Horizontal slices, average jumps per result at the point of intersection between the running time of the SRS and the kd-tree

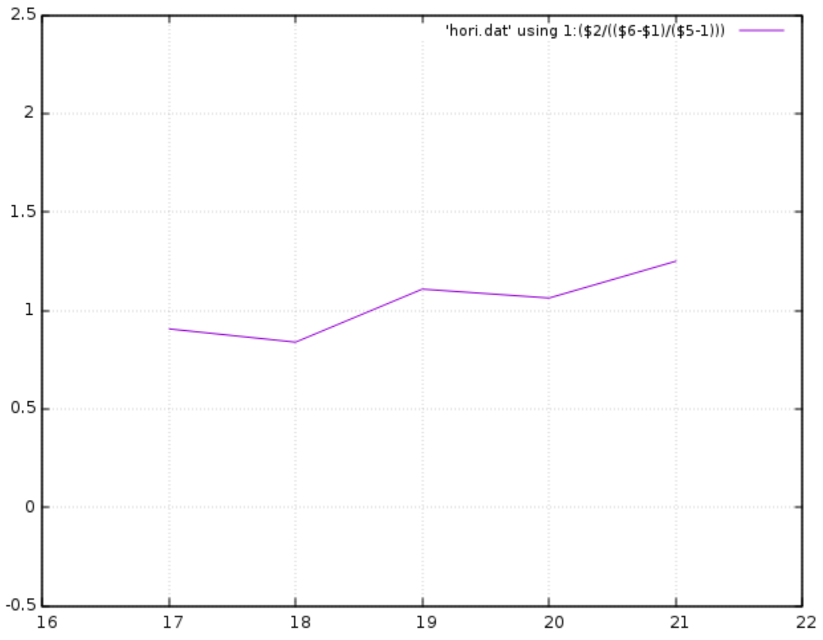

Figure 5.25: Horizontal slices, intersection between running time of SRS and kd-tree normalized by $k_{theoretical}$ with $\lg^{\epsilon} n$ being the average amount of jumps per result

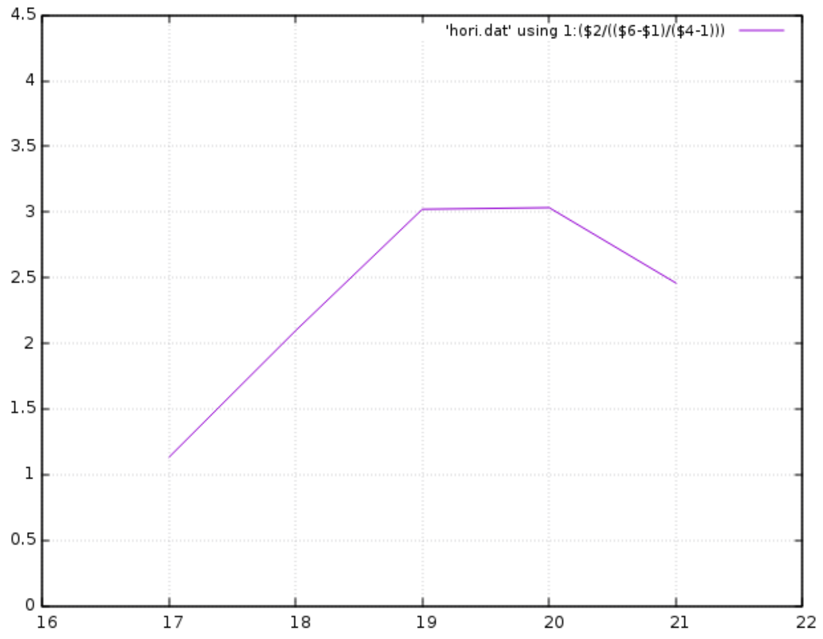

Figure 5.26: Horizontal slices, intersection between running time of SRS and kd-tree normalized by $k_{theoretical}$ with $\lg^{\epsilon} n$ being the average amount of jumps per result

Chapter 6

▶. . . ◀

Chapter 7

Conclusion

►. . .◄

Bibliography

[1] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008. ISBN 3540779736, 9783540779735.

[2] Timothy M. Chan, Kasper Green Larsen, and Mihai Pătraşcu. Orthogonal range searching on the ram, revisited. In *Proceedings of the Twenty-seventh Annual Symposium on Computational Geometry*, SoCG '11, pages 1–10, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0682-9. doi: 10.1145/1998196.1998198. URL `http://doi.acm.org/10.1145/1998196.1998198`.

[3] Johannes Fischer. Optimal succinctness for range minimum queries. *CoRR*, abs/0812.2775, 2008. URL `http://arxiv.org/abs/0812.2775`.

[4] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424 – 436, 1993. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/0022-0000(93)90040-4. URL `http://www.sciencedirect.com/science/article/pii/0022000093900404`.

[5] Roberto Grossi, Alessio Orlandi, Rajeev Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. *CoRR*, abs/0902.2648, 2009. URL `http://arxiv.org/abs/0902.2648`.