2D Orthogonal Range Search

Mads Ravn, 20071580

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

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Abstract

►in English. . . ◄

Resumé

►in Danish. . . ◄

Acknowledgements

►. . . ◄

Mads Ravn,
Aarhus, March 2, 2015.

Contents

Chapter 1

Introduction

►**Describe primary work**◄ In this thesis we are going to explore data structure we believe to be faster at orthogonal range search than a kd-tree. We are going to introduce the a range reporting data structure by Chan et al. [2]. A simplified version of their data structure will be made and implemented. We wish to show that this simplified version is able to compete with a kd-tree.

Orthogonal Range Searching. *Orthogonal range searcing* 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 k points lie within a given query range $\mathbb{Q} \subseteq \mathbb{R}^2$. This query can be defined as two corners of a rectangle, the lower left corner and the upper right corner, seeing as the query range is orthogonal to the axes.

RAM, I/O, pointer models ►**Skriv om Word RAM Model sammenlignet med de andre typer her. Når vi pakker bits ned i et word kan vi stadig tilgå det hele da access er constant time og operations er constant time**◄

Rank Space Reduction. Given n points from a universe U, the rank of a given point in a sorted list 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 for the *x-rank* r_x for each point in P by finding the rank of the x-coordinate amongst all the x-coordinates in P. The *y-rank* r_y finds the rank of y-coordinate amongst 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 $q^* = [r_x(x_1), r_x(x_2)] \times [r_y(y_1), r_y(y_2)]$. ►**Noget med at deres ordered property remains intact.**◄ 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. ►**We have essentially created a mapping between . . .**◄

Ball Inheritance. Given a perfect binary tree with n leaves and n labelled balls at the root, the goal is to distribute the balls from the root to the leaves. The balls at the root are contained in an ordered list and for each ball in a node's list, one of its children is picked to inherit the ball such that both children recieve the same amount of balls. The level of a node is defined to be the height of the node from the bottom. 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 that node's list. Given the identity of a ball at any level, it is possible to follow this identity to its actual coordinates. 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, where the actual coordinates reside, the *true identity* of a ball. When refering to ball inheritance, the words *ball* or *point* can be used interchangebly.►**Formatering**◄

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*. 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

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

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. $A[1..n]$ 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 first number which is greater or equal to x in that set - the same applies for predecessor of 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

In this chapter a simplification of the work done by Chan et al. [2] is presented followed by their original data structure. The simplification is the primary work of this thesis. Conceptually, the original structure by Chan et al. [2] can be thought of as an extension of the simplified data structure. This way the reader will be introduced to the concepts at an incremental level.►**rephrase**◄ The theory behind a kD-tree will also be explained as it is the de facto standard of orthogonal range queries today and will be used to compare the practical results of the simplified data structure. ►**rewrite**◄ Going forward, we will use *ORS* as shorthand for *Original Range Search* and *SRS* as shorthand for *Simplified Range Search*.

The SRS data structure has a time complexity of $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$ which is greather than the time complexity of the ORS data structure with $\mathcal{O}(\lg \lg n + (1+k) \cdot \lg^{e} psilonn)$. However, the SRS data structure is far more simple - both in code and the data structures used. Because there is much more internal communication between the data structures in the ORS data structure and much more code to be executed, it is safe to assume that the running time constant of ORS far exceeds that of SRS, making the SRS data structure faster than the ORS data structure in practise. ►**rephrase**◄

Some sections will start with a *preliminaries* subsection. This subsection will describe some of the auxiliary data structures used in the section. It will make the reader acquinted with the data structures when they are referenced.

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 is kd-trees. 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.

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 will be divided in a similar fashion, this time using the y-median and the y-coordinates instead. When dividing n points, the median will be chosen as the $\lceil n/2 \rceil$-th smallest number [1]. 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 takes up $\mathcal{O}(n)$ space.

In order to search in this tree, we introduce the term *region*. A region of a node is the area of which its point lie within. 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 narrow the region of both children. Doing this halves the amount of points lying within the region. Now given a search query $q = [x_1, x_2] \times [y_1, y_2]$ and a kd-tree one of three things can happen. The region of a node contain 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. The region of a node and the search query can overlap in which case we continue the search down both of the children of the node. Finally the region of a node and the search query has nothing in common in which case the search at that node stops. If a leaf is visited in the search, the point of the leaf is reported as part of the result if it lies within the search query.

Given a node, the time to visit and report the points stored in the subtree at the node is linear to the number of points reported. When the region of a node is fully contained in the search region it takes $\mathcal{O}(k)$ time to report k points as a result. It then takes $\mathcal{O}(k)$ time to report back all the k points which lie in the subtree of a node which is fully contained in the search region. In order to bound the case where the region of a node is not fully contained in the search region, a vertical line through the region of the root will be used. Conceptually, this vertical line can serve as either the left or right edge of the search region. Consider that the search region does not fully contain regions of any node, this line will describe the amount of regions the search query overlaps. $Q(n)$ will denote the amount of regions the vertical 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 other step. The vertical line will intersect two of these regions. The vertical line will also intersect the region of the root. When building the kd-tree the points are split by their x-coordinate, so the vertical line will only intersect the region of one of children of the root. Thus, the running time $Q(n)$ of a query to the kd-tree with n points can be described by the recurrence:

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

Solving this recurrence gives the solution $Q(n) = \mathcal{O}(\sqrt{n})$. A vertical line through the region of the root in a kd-tree will intersect $\sqrt{n}$ regions.

We can also imagine the regions of the kd-tree to form a $m \times m$ grid in the region of the root. Each region in the grid contains one point, and a vertical line through the grid will intersect m regions - one at each row. Since each region in the grid contains a point, we have $m \times m = n$ which gives $m = \sqrt{n}$. Thus, a vertical line through the region of the root will intersect $\sqrt{n}$ regions.

Searching the kd-tree takes $\mathcal{O}(\sqrt{n} + k)$ time to give 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 are linear to the amount of points.

The tree with n points can be represented as flat array with n entries. The $\lceil n/2 \rceil$-th element in the array is the root of the tree. **►uddyb◄**

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. **►rephrase!◄**

Consider a balanced binary search tree with n keys for a 1-dimensional query. 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 containing atleast x_1 and x_2, but 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 x_1. 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 has 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 to support range search on the y-axis of the points in 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 range of $[x_1, x_2]$ will be flagged. Using the auxilliry tree on each node that is flagged, a search will be done on to find the points in the range of $[y_1, y_2]$.

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 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 is then

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

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 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 from $A_v[i]$ 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.

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 chosen on the path to x_1 or x_2, the left pointer is used to update the position at τ. When a right child is chosen on the path, the position of τ is updated using the right pointer. 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 no longer is within the range of $[y_1, y_2]$. This takes $\mathcal{O}(1 + k_v)$ time. The total query time now becomes

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

2.3 Simplified Range Searching

This section will introduce the primary work of the thesis. **►this thesis eller the 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 $\mathcal{O}(\lg n + k \cdot \lg^\epsilon n)$ time. This is the same space complexity as the kd-tree. The query time is different in that $\lg n$ is smaller than $\sqrt{n}$, but there now is a penalty of $\mathcal{O}(\lg^\epsilon n)$ per point reported. Succint rank queries are an important part of this chapter, playing a key role in solving the ball-inheritance problem.

2.3.1 Preliminaries

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 ranks 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 find out how many elements in $A[1..i]$ are equal to $A[i]$. This is called a *rank query*. We want to be able to answer the 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 each checkpoint $\Sigma \lg n$ 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.

Furthermore, for each entry in the array A, a smaller sum will be stored. Each entry in A contains a character from the alphabet Σ. At $A[i]$ we store the amount of times the character at $A[i]$ appears 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 $x - 1$. Since there is only $\Sigma \lg n$ entries between each checkpoint, this adds up. 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. 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◄**

For smaller alphabets, another scheme is used. Checkpoinst are still stored at every $\Sigma \lg n$ positions. 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. 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◄**

2.3.2 Solving the ball-inheritance problem

Consider a perfect binary tree with n leaves. At each level a bit vector $A[1..n]$ is used to indicate which of a node's children a ball is inherited by: If $A[i]$ is 0 it means that the ball with identity i in that node was inherited by its left child and 1 means that it was inherited by its right child. 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 $rank_v(k) = \Sigma_{i \leq k} A[i]$. 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 which is represented by a bit vector of size n bits per level. Even though conceptually this bit vector is divided out amongst the nodes of that level, we can interchangebly think of it as a bit vector per level or a bit vector per node. 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.►**henvis til prelim**◄.

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 childrens children, $\Sigma = \{0, 1, 2, 3\}$, the childrens childrens 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.

Using this concept, we pick B such that $2 \leq B \leq m$, where m is the height of the tree. All levels that are a multiple of B^i expand their alphabet such that the balls 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 multiple of B^i costs B^i bits per ball. The total cost is then $\Sigma_i^{\lg_B \lg n} \frac{\lg n}{B^i} \cdot \mathcal{O}(B^i) = \mathcal{O}(\lg n \cdot \lg_B \lg n)$ 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 time complexity for solving the ball-inheritance is $\mathcal{O}(\lg^\epsilon n)$. ►**Det er fordi** $\lg_B \lg n$ **er det største** i **som beskriver** $B^i \leq \lg n$ ◄

2.3.3 Solving range reporting

Consider a perfect binary tree with n leaves. The root contains n points in 2-d rank space. These n balls 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 x-coordinates of the balls in the leaves are sorted from left to right - smallest to highest. 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 coordinate 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 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 $[y_1, y_2]$. Going forward we will use i_v and j_v to indicate 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 new 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 new 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 2.3.2. Now instead of just following a given ball, we keep track of a range of balls.

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 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 2.2.

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.

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 $\mathcal{O}(\lg n + k \cdot \lg^{\epsilon} n)$ query time to report k points as results. An empty range will be detected by the binary search at the root. If the binary search does not report an empty range, we proceed with the search. ►**Forklar hvordan** $\lg^{\epsilon} n$ **opstår fra "solving ball-inheritance**◄

2.4 Orthogonal Range Searching

Utilizing the ball-inheritance structure, Chan et al. [2] propose a better solution for orthogonal range search queries. Theorem 2.1 states that *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 $q = [x_1, x_2] \times [y_1, y_2]$.

2.4.1 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 refered 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 predessecor 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 smaller space complexity than the RMQ, but has 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 which has oracle access to the entries in the array. ►**reference**◄.

2.4.2 Solving range reporting

With a solution to the ball-inheritance problem, Chan et al. [2] proposes **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 2.3.2. 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 the 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 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. 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 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. **►rephrase◄**

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, $\hat{x}_1$, and the rank space predecessor of x_2, $\hat{x}_2$. 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 containing 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. Given that points are in an array, we can use xor as our LCA operation. **►rephrase◄**

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 $LCA(\hat{x}_1, \hat{x}_2)$. 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 $[\hat{y}_1, \hat{y}_2]$. Knowing the identity of a 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. 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 2.3.2. 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 LCA to the leaves solving the ball inheritance problem for the k results.

The three approaches described above all use a number of words linear to the amount of points. Theoretically, the ORS by Chan et al. [2] with its $\mathcal{O}(\lg \lg n + (1+k)(\lg^{\epsilon} n))$ query time is faster than the SRS. The kd-tree has no time penalty per point reported, while both the ORS and the SRS has a time penalty of $\mathcal{O}(\lg^{\epsilon} n)$ per point reported. The $\sqrt{n}$ in the kd-tree query time comes from the rather pessimistic view that a search region intersects the entire region of the root node, but does not fully contain any smaller region. Thus, the query time of the kd-tree is very reliant on how the search region intersects with the subdivived regions in the kd-tree. The query time of the kd-tree can therefore vary a lot, while the query time of the SRS will be more stable. While n grows, $\sqrt{n}$ grows at a much bigger rate than $\lg n$. There will be cases the kd-tree is better than the SRS and vice versa. **►Eksempler - alle punkter inkluderet og en meget tynd streg igennem◄**

►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?◄

►ADD CONSTANT TIME DIFFERENCE TALK◄

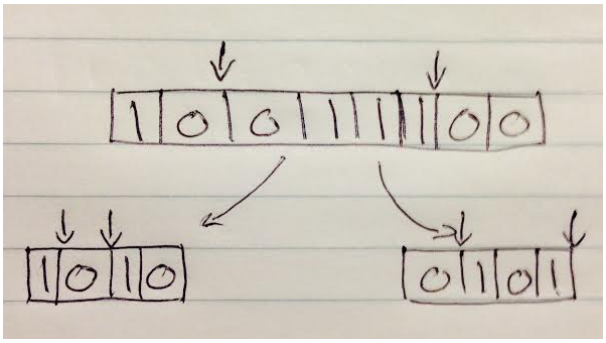

Figure 2.1: When nodes inherit their bit vector ranges from their parent, it can become obvious if the entire subtree is contained within the range of $[y_1, y_2]$ or falls without the range.

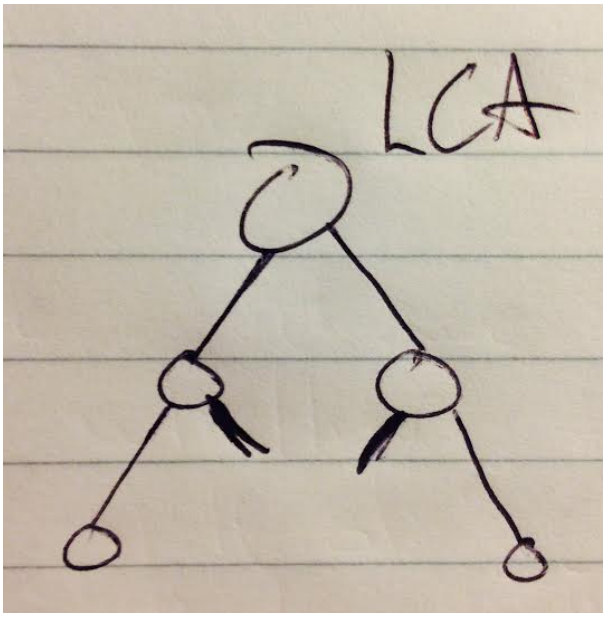

Figure 2.2: Traversing left from the LCA, each right subtree contains x-coordinates between x_1 and x_2. Traversing right from the LCA the same holds for left subtrees.

Chapter 3

►. . . ◄

Chapter 4

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`.