Zum Einstieg die Demo "Logics Operations –", falls noch nicht gezeigt

Image Compression

[J. Teuhola: "Fast Image Compression by Quadtree Prediction", 1998

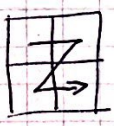
Anwendung: z.B. Bildübertragung von Rammsonden;
nur wenige hundert Bit/sec Übertr.geschw.;
langsame CPUs wegen Störstrahlung.

Ziel: einfacher und billiger Algo;
gleiche Bildqualität wie JPEG bei gleicher Datenrate (Bits/Pixel)

Ansatz: siehe Folien

Methode:

1. Vollständigen Quadtree bottom-up aufbauen;
dabei Min., Max. und Summe der Pixel-Werte nach oben propagieren
2. Quadtree beschneiden: Teilbaum abschneiden (d.h. Knoten zu Blatt machen) $\Leftrightarrow$
$\text{Max}(\text{Block}) - \text{Min}(\text{Block}) \leq \Theta$
Θ: User-definierter Schwellwert
3. Jedes Blatt durch
1-Grauwert repräsentieren: Mittelwert der Farbe $\frac{1}{2}(\text{Min}+\text{Max})$ oder $\frac{1}{n}\cdot$ Summe
4. Blatt-Grauwerte kodieren:
a) Quadtree mit DFS in z-Order durchlaufen!



b) Sei a = Grauwert $\in [0, 255]$
p = Wert eines Prädiktors
s = Seitenlänge des Blattes

Dann setze als kodierten Grauwert

$$code(p, a, s, \Theta) = \text{round}\left(\frac{a-p}{step(s,\Theta)}\right) = \begin{cases} \left[\left(\frac{a-p}{\Theta}\right)\cdot s\right] & \text{if } s<\Theta \\ [a-p], & \text{if } s>\Theta \end{cases}$$

mit

$$step(s,\Theta) = \max\left(1, \frac{\Theta}{s}\right)$$

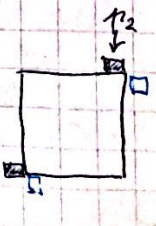
Falls $s \leq \Theta$: $code = \frac{(a-p)s}{\Theta}$
Falls $s > \Theta$: $code = (a-p)$

Beobachtung: $code \in [-255, 255]$, Häufung um 0.

Prädiktor hier:

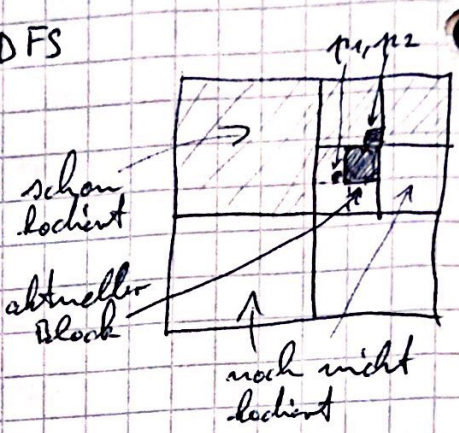
$$p = \frac{1}{2}(p_1 + p_2)$$

p_1 = kodierte (!) Werte der NE- / SW-Pixel

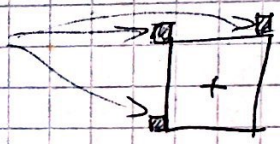


Bemerkung zum Prädiktor:

- p_1, p_2 kennt man immer, wegen z-Order-DFS
- müssen unbedingt die kodierten Werte sein, denn nur die hat man im Dekoder! (und der muss dasselbe ja auch machen!)



- p_1, p_2 sind i.A. keine Pixel-großen Blätter, sondern Teil eines größeren Blattes
- gibt noch viele weitere Möglichkeiten; meine Idee z.B.: verwende diese Pixel und lege eine Ebene da durch und werte diese in der Mitte des Blockes aus



- verwende für Blöcke am Rand nur 1 Prädiktor-Pixel
- verwende für den LO Block 128 (0(2)) als Prädiktor-Wert
- GPU? Multi-Layer Grid?

5. Bit-Kodierung: o.r.u. ⟶

a) Die Quadtree-Struktur: DFS in z-Order, (weil bei Treecode weiter sowieso) output schreibe 1 bei innerem Knoten, 0 bei Blatt
(Blätter auf Pixel-Ebene können noch eingespart werden)

b) Gewerte-Codes (schon quantisiert): z.B.

$0 \to 0$

$-1 \to 100$ $\quad 1 \to 101$
$-2 \to 1100$ $\quad 2 \to 1101$
$-3 \to 11100$ $\quad 3 \to 11101$

(also eine Art unäre Repräsentierung/Kodierung)

Rationale zu Schritt 4:

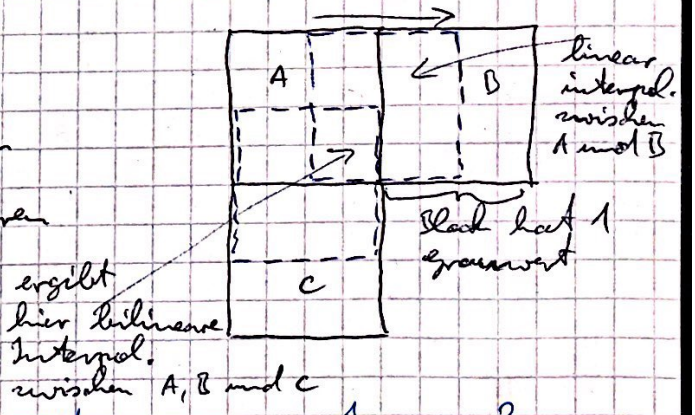
- Block liegt zwischen den beiden Präzisionsstufen
  → Grauwert sollte auch dazwischen liegen
- Quantisierung (durch Round()):
  ist hier eine variable-scale Quantisierung,
  wegen $\frac{1}{step(s,0)}$;
  je größer der Block, desto mehr Präzision;
  je größer die Variation, desto kleiner die Präzision
  (dann ist der exakte Pixelwert nicht so wichtig).
  Big block(s) means variation (max-min) is rel. small.
  If variation in area is big → Blocks will be small in that area → code small, because exact value not important

Dekompression:

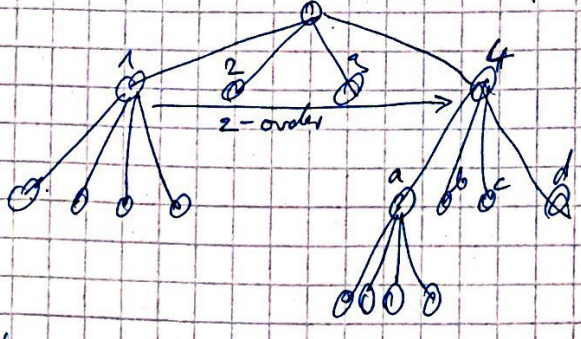
1. Quadtree aus Treecode aufbauen
2. Grauwerte rekonstruieren:

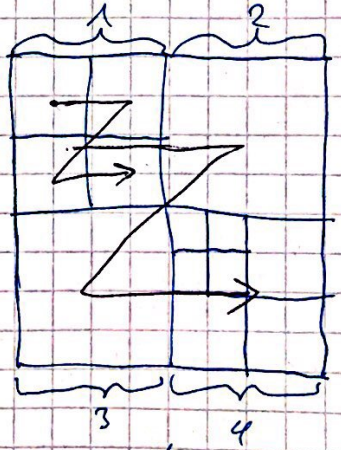
   a = code * step + p
   ↳ kennt man auch schon (fragment wie oben)

3. Block-Artefakte glätten: hier z.B. einfach zwischen benachbarten Blöcken linear die Pixel interpolieren



5.a) Generate Treecode for quadtree:





Output: 1 for inner node, 0 for leaf → DFS in z-order

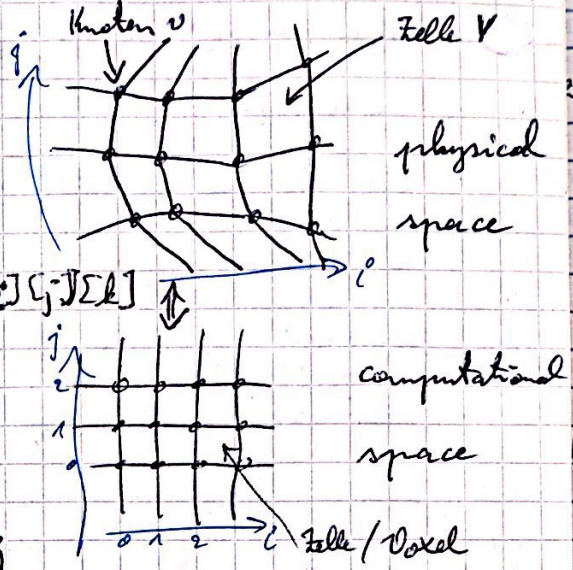
1 1 0 0 0 0 0 1 1 0 0 0 0 0 0 0

after each 0, output code for grey value of that leaf

0. Skalarfield
1. Erst Isosurface über stetig Fkt def.
2. Gitter einführen
3. Isosurface über Gitter definiere

Isosurface – Extraction:

Gegeben: kurvilineares Gitter mit Skalar $f(v)$ an jedem Gitter-knotenpunkt v;



implementiert durch 3D-Array $F[i][j][k]$ ← $f(r_{ijk}) \in \mathbb{R}$
mit Pkt + Skalar pro Eintrag.
↑ $r_{ijk} \in \mathbb{R}^3$ im physikal space

Isosurface A_τ zum Isowert τ:
($f: \mathbb{R}^3 \to \mathbb{R}$ ist im ganzen Raum def.)
kontinuierlich: $A_\tau = \{ x \in \mathbb{R}^3 \mid f(x) = \tau \}$
(f ist nur an den Knoten def.)
diskret: Eine Fläche A_τ, so daß für alle (manchmal auch "level set")
Voxel V, die A_τ schneiden, gilt

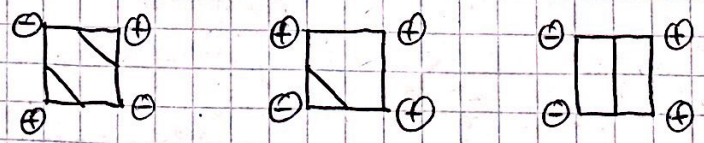
$$\exists v_i : f(v_i) \leq \tau \;\wedge\; \exists v_j : f(v_j) > \tau ,$$

wobei $\{v_1, \ldots, v_8\}$ die Knoten von V sind.

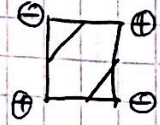
Diese Voxel kann man auch im computational space finden → ab jetzt nur noch dort arbeiten.

Einfachster Algo: Marching Cubes (1987)

laufe (linear) durch Voxel-Array;
bestimme Vorzeichen der Ecken des Voxels ($\oplus \hat{=} > \tau$, $\ominus \hat{=} \ldots$)
trianguliere Voxel gemäß LUT



aber



In 3D gibt es 2^8 Fälle
(=15 modulo Spiegelungen (Rotation)

↖ Fall einer Nicht-Eindeutigkeit;
gibt im 3D noch ein paar mehr.

Nicht-triviał, die erschöpfende Suche schneller zu machen, denn man muß alle Komponenten der Isosurface entdecken.

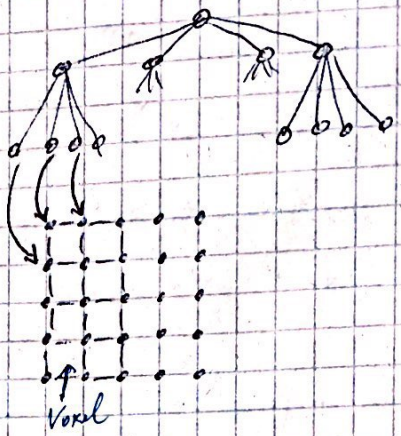
→ Octree. Min-Max-Octree

Voller Octree;

Blätter zeigen auf Voxel, z.B. li.-untere Knoten $F[i][j][k]$ Ecke der zugehörigen Zelle.

Blätter haben min/max der 8 Knoten.

Innere Knoten haben min/max der 8 Kinder.



Wilhelms & van Gelder. "Octrees for Faster Isosurface Generation"

Beob.:

Isosurface läuft durch Voxelbereich eines Octree-Knoten v

$\Leftrightarrow \min(v) \leq \tau \leq \max(v)$.

→ Algo. (Ü-aufgabe)

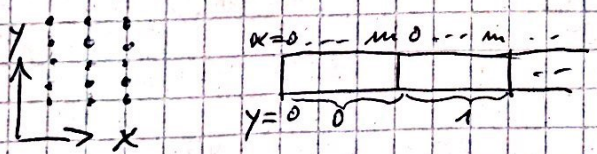
Optimierung: Memorization der Vertices der Isosurface; Beob.: alle Vertices liegen auf Gitterkanten
→ Hash-Table für Kanten des Voxel-Gitters:

speichert Zeiger auf Pkt in Vertex-Array plus Normale; Vertex on edge

wann? → Eintrag löschen, wenn Kante zum letzten Mal besucht

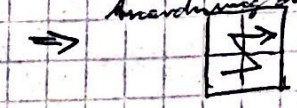
(fast alle K. werden genau 4x besucht, außer am Rand.)

Optimierung: Octree-Layout / -Organisation soll zum Layout des Volumendatensatzes passen:

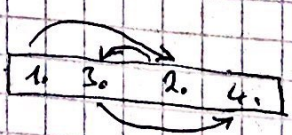


"row-major"

→ Anordnung der Kinder eines Octree-Knotens minimiert Page-Faults



würde im Speicher "springen":



Exkurs: Isoflächen von zeit-veränderlichen Feldern:

~~Exkurs~~ "Exkurs Exkurs" zuerst !! (2 Blätter weiter)

Gegeben: N 3-dim. Skalarfelder, jedes beschreibt Zustand zur Zeit $t_i \in [t_0, \ldots, t_{N-1}]$

(Schreibe ab jetzt $t=i$ statt t_i.)

[Shen: Isosurface Extraction in Time-Varying Fields Using a Temporal Hierarchical Index Tree. Visualization '98.]

Triviale Lsg.:
bäume Octree für jeden Zeitschritt.
Problem: braucht zu viel Speicher

Gesucht: Datenstrukturen, die weniger Speicher braucht, und trotzdem schnelle Isoflächen ermöglicht.

Def.:

$\min_t / \max_t(v) :=$
Min./Max. über die 8 Knoten einer Zelle v in Zeitschritt t. [analog zu bei Octree]

"Span Space" :=
Ebene, in der Zellen v als Punkte aufgetragen werden, mit $\min(v) \hat{=} x_v$, $\max(v) \hat{=} y_v$
Zu geg. θ liegen alle Zellen mit $\min < \theta < \max$ "links oben" von (θ, θ).

"zeitliches Min./Max." einer Zelle v :=
$\min_i^j(v) := \min_{t \in [i,j]} \{ \min_t(v) \}$, $\max_i^j := \ldots$

Beob.: t laufen lassen → $(\min_t(v), \max_t(v))$ einer Zelle v wandert im Span-Space; je weiter (lt) wandert, desto größer ist zeitliche Variation der Zelle.

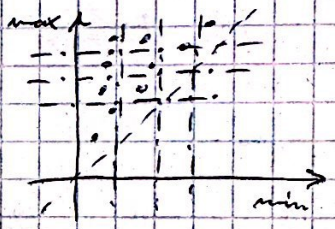
Wunsch: je kleiner zeitliche Variation einer Zelle, desto mehr t's will man zusammenfassen.

Lsg.: Span-Space

Gegeben: Zeitintervall $[i,j]$ u. Menge von Zellen V;

gesucht: Zellen mit "kleiner" zeitlicher Variation.

Erzeuge Plotte alle Punkte $(\min_t, \max_t)$ aller Zellen $v \in V$ für alle $t \in [i,j]$ im Span-Space;



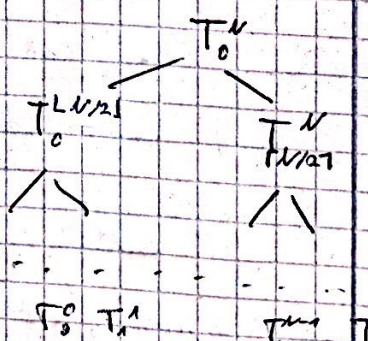
unterteile Span-Space in $L \times L$ Lattice-Zellen, so daß in jeder Zeile/Spalte ungefähr gleich viele Punkte sind.

Def.: Zelle $v \in V$ hat "kleine zeitliche Variation" über Zeitintervall $[i,j]$:⇔ alle Punkte $(\min_t(v), \max_t(v))$, $t = i \ldots j$, liegen in höchstens 2×2 zusammenhängenden Lattice-Zellen.

Temporal index tree (TI tree)

Def. Zeitlicher Index-Baum (ZIB):

T_i^j enthält alle diejenigen $v \in V$, die eine kleine zeitliche Variation über Zeitintervall $[i,j]$ haben, und die nicht schon weiter oben enthalten sind.



$V(T_i^j) := \{ v \in V \mid v$ hat kleine zeitliche Variation über $[i,j] \}$

Aufbau des zeitlichen Index-Baumes:

Starte mit V und T_0^N;

baue Lattice über Span-Space von V und Intervall $[0,N]$;

checke für jedes $v \in V$ Bedingung für "kleine zeitliche Variation über Intervall $[0,N]$";

falls erfüllt → füge v zu $V(T_0^N)$ hinzu;

Rekursion mit $T_0^{\lfloor N/2 \rfloor}$, $T_{\lceil N/2 \rceil}^N$, und $V \setminus V(T_0^N)$;

→

Klar: alle Zellen erfüllen Bedingung für "kleine zeitl. Var." über Intervall $[t,t]$, d.h., für einen Zeitschritt.

→ Zellen mit sehr großer zeitl. Variation landen in den Knoten T_i^i, $i = 0..N$, also N Mal.

(ist aber nicht schlechter als bei getrennten Octrees.)

~~Notwendig~~ Hinreichende Bedingung für Isosurface-Zellen:

Frage: notwendige od. hinreichende Bed.? → notw.

$$\min_i^i(v) < \theta < \max_i^i(v) \Rightarrow$$

$$\exists t_1: \min_{t_1}(v) < \theta \quad \wedge \quad \exists t_2: \max_{t_2}(v) > \theta.$$

≙ Voxel über mehr. Zeitsch.

baue für alle Voxel $(\min_i^i(v), \max_i^i(v)) \in V(T_i^i)$ einen Octree (no longer complete octree) wie für statische Isosurface-Generierung.

~~Octree ist natürlich schlecht weil, da in jedem T_i^i nur ein Bruchteil aller Zellen gespeichert~~

Isosurface-Generierung:

gegeben: t, θ

(1) traversiere zeitlichen Index-Baum von T_0^N bis T_t^t, d.h., für alle Knoten T_a^b auf dem Pfad gilt: $a \le t \le b$;

(2) für jeden Knoten T_a^b (auf dem Pfad) mache mittels Octree Zellen v, für die $\min_a^b(v) < \theta < \max_a^b(v)$ (wie bei statischer Isosurface)

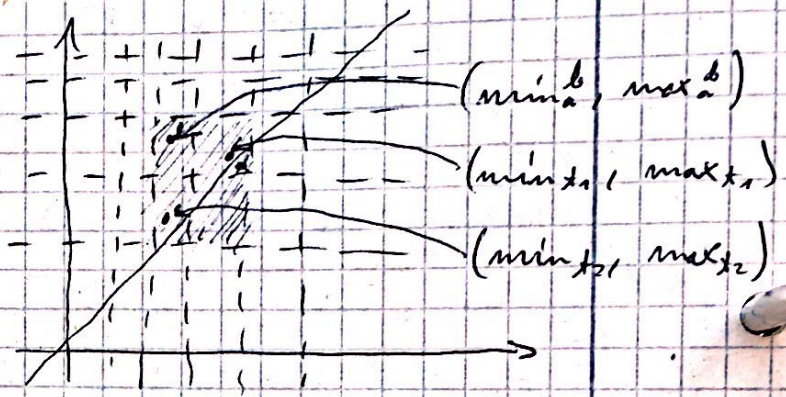
(3) filtere diese Zellen, d.h. teste ob $\min_t(v) < \theta < \max_t(v)$

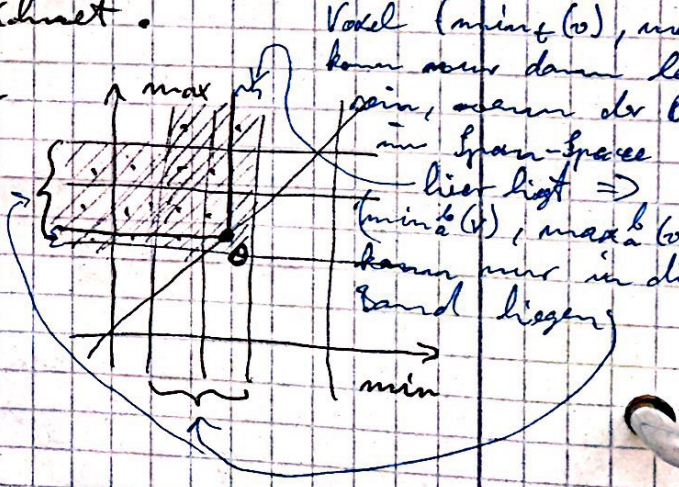
Bem.: 1. alle Zellen kommen auf dem Pfad $T_0^N \to T_t^t$ genau 1× vor.

Bem.: 2. Im Schritt (2) kommen evtl. auch Zellen =Voxel raus, die nicht zur Isosurface gehören; ~~werden bei Triangulierung gemäß Marching-Cubes-Tabelle gefiltert.~~

Wie viele "leere" Voxel-Zellen werden besucht:

1. In Z-Blättern werden durch Octree nur Voxel-Zellen geliefert mit $min_t < \theta < max_t$, die also auf Isosurface liegen müssen.
2. Licht-Blätter: Octree liefert Voxel-Zellen mit $min_a^b < \theta < max_a^b$;
   vgl. Variationsbeschränkung
   müssen alle Ekte $(min_t, max_t)^{t=a..b}$, dieser Voxel-Zelle in einem 2x2-Lattice-Zellen-Bereich liegen, der (min_a^b, max_a^b) enthält;
   worst-case im Bild ↗ eingezeichnet.
   ⇒ nur Voxel-Zellen, deren (min_a^b, max_a^b) im dem grau schattierten Bereich liegt, können zur Zeit $t \in [a,b]$ "leer" sein.





→ Mit L kann man zwischen Speicheraufwand und Geschw. wählen.

Out-of-Core-Visualisierung:
Nicht alle Datensätze passen in Speicher;
mmap() mapt File auf virtuellen Adressraum;
hat man eine Isosurface zur Zeit t, dann sind durch den zeitlichen Index-Tree schon viele Zellen im Speicher, die für Isosurface zur Zeit $t+1$ gebraucht werden (und zwar um so wahrscheinlicher, je weiter oben).

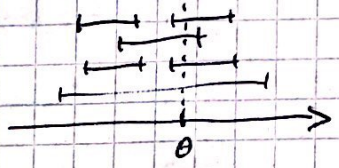
Exkurs Exkurs: Der Span Space

(Manchmal gibt es nicht-hierarchische Datenstrukturen, die schneller ein Problem lösen können als die hierarchischen – zumindest in praktischen Fällen.)

Problem "1-dim stabbing query":

gegeben: N Intervalle $[a_i, b_i] \subseteq \mathbb{R}$

θ = Anfragepkt

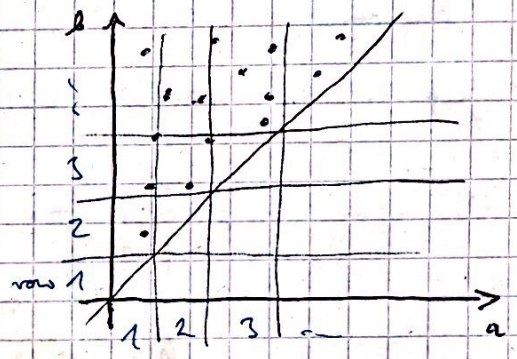


gesucht: alle Intervalle mit $\theta \in [a_i, b_i]$

Standardalgo: Interval-Tree oder Segment-Tree

Man kann die Konstante drücken!

Idee: verwende Span-Space.
Consider $[a,b]$ as points $(a,b) \in \mathbb{R}^2$.
Lege Lattice über Span-Space,



so daß

1. Zkte zeilenweise u. spaltenweise ungefähr gleich verteilt sind,
2. Linienabstände auf a- und b-Achse sind gleich

(2) ⇒ Lattice-Zellen werden von der Diagonale nicht geschnitten, oder genau diagonal.

(1) macht man durch Sortieren aller a,b-Wte auf einer Achse.

Für jeder von Zeile i des Lattice speichere 2 Listen:

1. L_i^a enthält alle Zkte der Zeile von Spalte 1 ... $(i-1)$, aufsteigend sortiert nach a
2. L_i^b ..., absteigend sortiert nach b

Für den Lattice-Zellen auf der Diagonale, evtl. trivially, if not too many on diag.
erzeuge wieder Lattice (bis Rek.-Tiefe d = const) oder Interval-Tree.

Algo:

finde Lattice-Zelle (l, l) die (Θ, Θ) enthält;

für Zeilen $i = l+1, \ldots$:

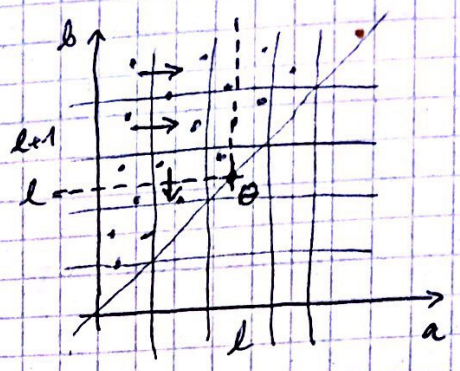
traversiere Liste L_i^a bis $L_i^a[j] > \Theta$

für Zeile l:

traversiere Liste L_l^b bis $L_l^b[j] < \Theta$

für Zelle (l, l):

suche im Lattice dieser Zelle, oder exhaustive search aber im Intervall-Tree dort.



Aufwand:

Jede l.-Zelle enthält ca. $\frac{2n}{L^2}$ Pkte

$$T(n) = O(l) + O(L) + O(\log L) + T\left(\frac{2n}{L^2}\right)$$

$= O(\log \frac{n}{L^2})$ im Falle interne Intervall tree für diagonale Zelle

Frage: wieso "2"? $\rightarrow \frac{n}{L^2/2}$ viele Pkte

↑ auf der Diag.-Zelle

↑ Finden von l mittels Binärsuche

↑ Für erste Schleife (sehr kleine Konstante, da Schleifenrumpf schon im $O(k)$ steckt)

↑ Anzahl Diagonale-Pkte

Leider: wenn man Rekursion der Lattice-DS so lange macht, bis Diag.-Elemente nur noch 1 Pkt enthalten, dann bekommt man Aufwand

$$O\left(L \cdot \log_{L^2/2} n + k\right).$$ (Intervall tree hat $O(\log n + k)$)

Ergibt z.B. $O\left(\frac{\log^2 n}{\log \log n}\right)$ für $L = \log n$, was schlechter als $O(\log n)$ ist.

Prakt. Optimierung: speichere L_i^a "zellenweise", j

dann muß man in der ersten Schleife des Algos oben nur die Zelle in Spalte l checken – alle Punkte darvor kann man direkt unbesehen ausgeben.