Advanced Data Structures

Lecture 06: Orthogonal Range Searching and BSP Trees
Florian Kurpicz and Stefan Walzer
Motivation: Query Set of Points

- given set of points \( P = \{p_1, \ldots, p_n\} \) with \( p_i = (x_i, y_i) \)
- find all points in \([x, y] \times [x', y']\)
- higher dimensions are possible

- think about database queries
- each dimension is a property
- searching for objects fulfilling all properties of range
consider 1-dimensional problem
range is \([x..x']\)
points \(P = \{x_1, \ldots, x_n\}\) are just numbers

build BBST where each leaf contains a point
inner node \(v\) store splitting value \(x_v\)

query for both \(x\) and \(x'\)
find leaves \(b\) and \(e\) for \(x\) and \(x'\)
let node \(v\) be node where paths to leaves split
report all leaves between \(b\) and \(e\)
how long does it take to report all children of a subtree with \( k \) leaves in a BBST?

**Lemma: 1-Dimensional Range Searching**

Let \( P \) be a set of \( n \) 1-dimensional points. \( P \) can be stored in a BBST that requires \( O(n) \) words space, can be constructed in \( O(n \log n) \) time, and can answer range searching queries in \( O(\log n + occ) \) time.

**Proof (Sketch Time)**

- reporting all children in a subtree requires \( O(occ) \) time
- BBST has depth \( O(\log n) \)
- search paths starting at \( v \) have length \( O(\log n) \)
- report all subtrees to the right of the left path
- report all subtrees to the left of the right path
2-Dimensional Rectangular Range Searching

Important

- assume no two points have the same x- or y-coordinate ⇒ general position

Generalize 1-dimensional idea

1-dimensional
- split number of points in half at each node
- points consist of one value

2-dimensional
- points consist of two values
- split number of points in half w.r.t. one value
- switch between values depending on depth
Kd-Trees (1/4)

- considering the 2-dimensional case
- each inner node at an even depth
  - splits the leaves in its subtree in half
  - using the x-coordinate
- each inner node at an odd depth
  - splits the leaves in its subtree in half
  - using the y-coordinate
- until each region contains a single point
- each leaf represents a point

- splitting in linear time is complicated
- better presort based on x- and y-coordinate
- inner nodes store splitter (line)
Kd-Trees (2/4)

Lemma: Kd-Tree Construction

A kd-tree for a set of $n$ points requires $O(n)$ words space and can be constructed in $O(n \log n)$ time.

Proof (Sketch: Space)
- there are $O(n)$ leaves
- there are $O(n)$ inner nodes
- a binary tree requires $O(1)$ words per node
- $O(n)$ words total space

Proof (Sketch: Time)
- finding the splitter is easy due to presorted points
- splitting requires $T(n)$ time with
  $$T(n) = \begin{cases} O(1) & n = 1 \\ O(n) + 2T(\lceil n/2 \rceil) & n > 1 \end{cases}$$
- results in $O(n \log n)$ running time
- presorting in same time bound
use splitter depending on depth to identify paths through tree

- if a region is fully contained in query: report region
- if a region is intersected by query: check if point has to be reported

precomputation of query not necessary
- current region can be computed during query
- using splitters

example on the board
Lemma: Kd-Tree Query
A query with an axis-parallel rectangle in a Kd-tree storing \(n\) points in the plane can be performed in \(O(\sqrt{n} + \text{occ})\) time

Proof (Sketch)
- \(O(\text{occ})\) time necessary to report points
- look at number of regions intersected by any vertical line
- upper bound for the regions intersected by query (for left and right edge of rectangle)
- upper bound for top and bottom edges are the same

Proof (Sketch, cnt.)
- for vertical lines consider every inner node at odd depth
- starting at root’s children
- can intersect two regions
- number of nodes is \([[n/4]\) halved at each level
- number of intersected regions is \(Q(n)\) with

\[
Q(n) = \begin{cases} 
O(1) & n = 1 \\
2 + 2Q(\lceil n/4 \rceil) & n > 1 
\end{cases}
\]

- results in \(Q(n) = O(\sqrt{n})\)
- \(O(\sqrt{n} + k)\) total running time
Teaser: Other Space-Partitioning Search Trees

- **Quadtrees**
  - recursive partition of input space into four children (top-down)
  - generalizes to higher dimensions (Octtree)
  - often used in computer graphics

- **R-Trees**
  - recursively group nearby objects into minimal bounding boxes (bottom-up)
  - works also for complex shapes, not only points
  - many variants exist (R*-Trees, R+Trees)
  - often used in spatial databases

Example on the board 🎨
Range Trees (1/4)

- one BBST build on the x-coordinates
  - same as for 1-dimensional queries
- each inner node is associated with a set of points
- build a BBST for the y-coordinates of associated points for each inner node
  - store points in leaves not just y-coordinates
  - this BBST is used for reporting
- space-query-time trade-off
- faster queries but larger
Range Trees (2/4)

- the BBST for the \( x \)-coordinates requires \( O(n) \) words of space
- how much space do the associated BBSTs require in total? PINGO

Lemma: Space Range Tree

A range tree on a set of \( n \) points in the plane requires \( O(n \log n) \) words space

Proof (Sketch)

- BBST for \( x \)-coordinates has depth \( O(\log n) \)
- all points are represented on each depth exactly once

Proof (Sketch, cnt.)

- all associated BBSTs on each depth contain every point exactly once
- total size of all BBSTs on each depth is \( O(n) \)
- total space \( O(n \log n) \) words

how much faster is the range tree?
2-dimensional rectangular range search reduced to two 1-dimensional range searches
look in BBST for $x$-coordinates same as 1-dimensional case
instead of reporting subtrees to the right/left of paths search associated BBSTs
report results in leaves of associated BBSTs

**Proof (Sketch)**
- each search in an associated BBST $t$ requires $O(\log n + occ_t)$ time
- $O(\log n)$ associated BSSTs $T$ are searched same as seen in 1-dimensional case
- total query time $\sum_{t \in T} O(\log n + occ_t) = O(occ)$
- $\sum_{t \in T} O(\log n) = O(\log^2 n)$
- total time: $O(\log^2 n + occ)$

**Lemma: Range Tree Query Time**
A query with an axis-parallel rectangle in a range tree storing $n$ points requires $O(\log^2 n + occ)$ time
Range Trees (4/4)

- range trees can be generalized to higher dimensions
- for each dimension add an additional associated BBST
- reporting in final BBST
- $d$-dimensional queries are $d$ 1-dimensional queries

Lemma: Higher Dimensions Range Tree

A $d$-dimensional range tree (for $d \geq 2$) storing $n$ points in the plane requires $O(n \log^{d-1} n)$ words space and can answer queries in $O(\log^d n + \text{occ})$ time.

Proof (Sketch Query Time)

- recursive query time $Q_d(n)$ with $Q_2(n) = O(\log^2 n)$
- $Q_d(n) = O(\log n) + O(\log n) \cdot Q_{d-1}(n)$
- solves to $Q_d(n) = O(\log^d n)$
- $O(\text{occ})$ time for reporting

Proof (Sketch Construction Space)

- recursive space $S_d(n)$ with $S_2(n) = O(n \log n)$ words
- $T_d(n) = O(n \log n) + O(\log n) \cdot T_{d-1}(n)$
- solves to $S_d(n) = O(n \log^{d-1} n)$
Fractional Cascading (1/2)

- sorted sets $S_1, \ldots, S_m$
- $|S_1| = n$ and $S_{i+1} \subseteq S_i$
- report elements in range $[x..x']$ in $S_1, \ldots, S_m$

- in addition to $S_i$ store pointers to $S_{i+1}$
- for each element in $S_i$ store pointer to successor in $S_{i+1}$
- possible because $S_{i+1} \subseteq S_i$

- how much time does a naive algorithm with binary search require? PINGO
- $O(m \log n + occ)$ time
- $O(m + \log n + occ)$ time possible with fractional cascading
Lemma: Fractional Cascading

Given sets $S_1, \ldots, S_m$ with $|S_1| = n$ and $S_{i+1} \subseteq S_i$, find a range in all $S_i$'s using fractional cascading requires $O(m + \log n + \text{occ})$ time.

Proof (Sketch)

- Binary search on $S_1$ requires $O(\log n)$ time.
- Following pointer to $S_2$ requires $O(1)$ time.
- Scanning $S_2$ requires $O(\text{occ})$ time.
- Following pointer to $S_3$ requires $O(1)$ time.
- Repeat $m$ times.
- Total: $O(m + \log n + \text{occ})$ time.

how to apply to range trees?
- Instead of associated BBSTs store leaf data in arrays for all nodes but root.
- Each node has associated data.
- Store two successor pointers to the associated data in the left and right child.
- Two pointers to cover all possible paths.
- This is a layered range tree.
Query Layered Range Trees

- search in BBST for x-coordinates remains the same
- to search y-coordinates first search associated BBST of root
- same as initial binary search for fractional cascading
- continue to follow pointers in associated data and scan to report queries

Proof (Sketch)

- the initial search requires $O(\log n)$ time
- the search in the associated BBST of the root requires $O(\log n)$ time
- remaining searches in associated data $a$ requires $O(1 + occ_a)$ time
- each point is reported once
- total time: $O(\log n + occ)$

Lemma: Query time Layered Range Tree

A query with an axis-parallel rectangle in a layered range tree storing $n$ points in the plane can be performed in $O(\log n + occ)$ time
all solutions requires unique $x$ and $y$-coordinates

big limitation for applications

remember database motivation

store $(x|k)$ as coordinate with $x$ being the $x$-coordinate and $k$ a unique key

same for $y$-coordinates

compare points using

$$(x|k) < (x'|k') \iff x < x' \text{ or } (x = x' \text{ and } k < k')$$

range queries $[x..x'] \times [y..y']$ become

$$[(x| - \infty) .. (x'|\infty)] \times (y| - \infty) .. [(y'|\infty)]$$
all solutions requires unique $x$ and $y$-coordinates

big limitation for applications

remember database motivation

if exact positions are not important to application

random perturbation: $x + \delta \sim U(-\epsilon, \epsilon)$

same for $y$-coordinates

range queries $[x..x'] \times [y..y']$ become

$$[(x - \epsilon) .. (x' + \epsilon)] \times (y - \epsilon) .. [(y' + \epsilon)]$$
Now: Render Object

- hidden surface removal
- which pixel is visible
- important for rendering
z-Buffer Algorithm

- Transform scene such that viewing direction is positive z-direction
- Consider objects in scene in arbitrary order
- Maintain two buffers
  - Frame buffer: currently shown pixel
  - Z-buffer: z-coordinate of object shown
- Compare z-coordinate of z-buffer and object
- First sort object in depth-order
- Depth-order may not always exist
- How to efficiently sort objects?
BSP Trees (1/2)

- partition space using hyperplanes
- binary partition similar to kd-tree
- hyperplanes create half-spaces and cut objects into fragments

- \( h^+ = \{(x_1, \ldots, x_d): a_1x_1 + \cdots + a_dx_d > c\} \)
- \( h^- = \{(x_1, \ldots, x_d): a_1x_1 + \cdots + a_dx_d < c\} \)

- each split creates two nodes in a tree
- if number of objects in space is one: leaf
- otherwise: inner node
BSP Trees (2/2)

- for leaf: store object/fragment
- for inner node $v$: store hyperplane $h_v$ and the objects contained in $h_v$
- left child represents objects in upper half-space $h^+$
- right child represents objects in lower half-space $h^-$

- space of BSP tree is number of objects stored at all nodes
- what about fragments?
- too many fragments can make the tree big
Auto-Partitioning

- sorting points for kd-trees worked well
- BSP-tree is used to sort objects in depth-order
- auto-partitioning uses splitters through objects
  - 2-dimensional: line through line segments
  - 3-dimensional: half-plane through polygons
Painter’s Algorithm

- consider view point $p_{\text{view}}$
- traverse through tree and always recurse on half-space that does not contain $p_{\text{view}}$ first
- then scan-convert object contained in node
- then recurse on half-space that contains $p_{\text{view}}$
Constructing Planar BSP Trees (1/3)

- use auto-partitioning
- construction similar to construction of kd-tree
- store all necessary information
  - hyperplane
  - objects in hyperplane
- how to determine next hyperplane?
- creating fragments increases size of BSP tree

- let s be object and \( \ell(s) \) line through object
- order matters
Lemma: Number Line Fragments

The expected number of fragments generated when iterating through the line segments using a random permutation is $O(n \log n)$

Proof (Sketch)

- distance of lines $\text{dist}_{s_i}(s_j) =$
  \[
  \begin{cases}
  \# \text{ segments inters. } \ell(s_i) \\
  \text{between } s_i \text{ and } s_j \\
  \infty \\
  \ell(s_i) \text{ inters. } s_j \\
  \text{otherwise}
  \end{cases}
  \]

- example on the board

Proof (Sketch, cnt.)

- let $\text{dist}_{s_i}(s_j) = k$ and $s_{j_1}, \ldots, s_{j_k}$ be segments between $s_i$ and $s_j$
- what is the probability that $\ell(s_i)$ cuts $s_j$?
- this happens if no $s_{j_x}$ is processed before $s_i$
- since order is random

\[
\mathbb{P}[\ell(s_i) \text{ cuts } s_j] \leq \frac{1}{\text{dist}_{s_i}(s_j) + 2}
\]
Proof (Sketch, cnt.)

- expected number of cuts

\[ \mathbb{E}[\# \text{ cuts generated by } s_i] \leq \sum_{j \neq i} \frac{1}{\text{dist}_{s_i}(s_j)} + 2 \leq 2 \sum_{k=0}^{n-2} \frac{1}{k+2} \leq 2 \ln n \]

- all lines generate at most \(2n \ln n\) fragments

Lemma: BSP Construction

A BSP tree of size \(O(n \log n)\) can be computed in expected time \(O(n^2 \log n)\)

Proof (Sketch)

- computing permutation in linear time
- construction is linear in number of fragments to be considered
- number of fragments in subtree is bounded by \(n\)
- number of recursions is \(n \log n\)
Conclusion and Outlook

This Lecture
- orthogonal range searching
- BSP trees

Advanced Data Structures

- Successor
- RMQ
- static BV
- static succ. trees
- range min-max tree
- succ. graphs
- Kd-/ Range / BSP Tree