Finding Out Where You Are: Simple and Fast on Average

David M. Mount

Department of Computer Science
University of Maryland, College Park

Joint with Sunil Arya and Charis Malamatos

REU Talk – 2021

Where in the heck am I?

Point Location Problem

- Given a subdivision of the plane into disjoint regions
- . . . preprocess it into a data structure, so that given any query point q
- . . . we can quickly determine which region contains q

Where in the heck am I?

Point Location Problem

- Given a subdivision of the plane into disjoint regions
- . . . preprocess it into a data structure, so that given any query point q
- . . . we can quickly determine which region contains q

Where in the heck am I?

Point Location Problem

- Given a subdivision of the plane into disjoint regions
- . . . preprocess it into a data structure, so that given any query point q
- . . . we can quickly determine which region contains q

Applications

Point location is used in:

- Geographic information systems: Which {country, state, county, city, school district} contains a given point?
- Nearest neighbor searching: What is the closest site (post office, hospital, gas station) to a given location?
  - Let P denote a set of point sites in the plane
  - Build the Voronoi diagram of the sites
  - The site defining q's cell is its nearest neighbor

But how do you find this cell efficiently?

Applications

Point location is used in:

- Geographic information systems: Which {country, state, county, city, school district} contains a given point?
- Nearest neighbor searching: What is the closest site (post office, hospital, gas station) to a given location?
  - Let P denote a set of point sites in the plane
  - Build the Voronoi diagram of the sites
  - The site defining q's cell is its nearest neighbor

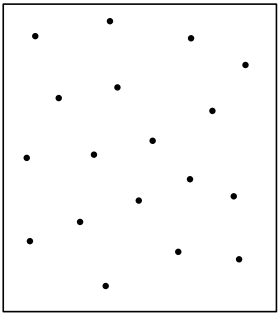
But how do you find this cell efficiently?

Applications

Point location is used in:

- Geographic information systems: Which {country, state, county, city, school district} contains a given point?
- Nearest neighbor searching: What is the closest site (post office, hospital, gas station) to a given location?
  - Let P denote a set of point sites in the plane
  - Build the Voronoi diagram of the sites
  - The site defining q's cell is its nearest neighbor

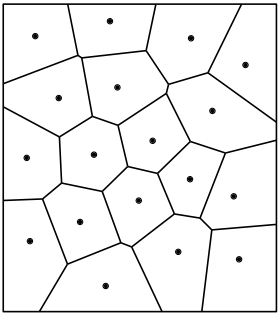
But how do you find this cell efficiently?



Applications

Point location is used in:

- Geographic information systems: Which {country, state, county, city, school district} contains a given point?
- Nearest neighbor searching: What is the closest site (post office, hospital, gas station) to a given location?
  - Let P denote a set of point sites in the plane
  - Build the Voronoi diagram of the sites
  - The site defining q's cell is its nearest neighbor

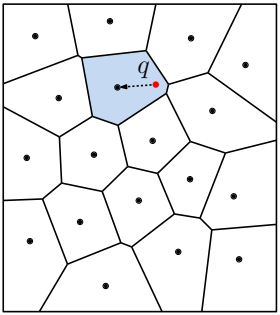


But how do you find this cell efficiently?

Applications

Point location is used in:

- Geographic information systems: Which {country, state, county, city, school district} contains a given point?
- Nearest neighbor searching: What is the closest site (post office, hospital, gas station) to a given location?
  - Let P denote a set of point sites in the plane
  - Build the Voronoi diagram of the sites
  - The site defining q's cell is its nearest neighbor

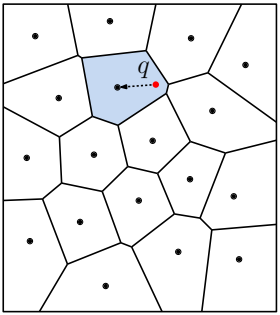


But how do you find this cell efficiently?

Applications

Point location is used in:

- Geographic information systems: Which {country, state, county, city, school district} contains a given point?
- Nearest neighbor searching: What is the closest site (post office, hospital, gas station) to a given location?
  - Let P denote a set of point sites in the plane
  - Build the Voronoi diagram of the sites
  - The site defining q's cell is its nearest neighbor



But how do you find this cell efficiently?

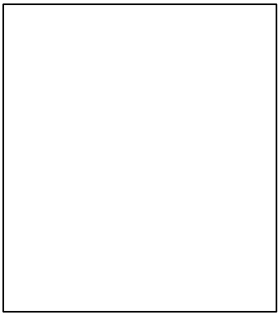
Efficiency

How do we measure efficiency?

- Space: How much storage is needed to store the data structure?
- Query Time: How much time is needed to answer a query?

Let n denote the total space needed to store the map: vertices, edges, faces

Gold Standard (Worst-case): $O(n)$ space and $O(\log n)$ query time.



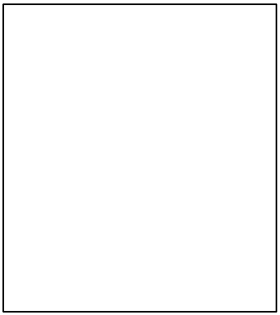
Efficiency

How do we measure efficiency?

- Space: How much storage is needed to store the data structure?
- Query Time: How much time is needed to answer a query?

Let n denote the total space needed to store the map: vertices, edges, faces

Gold Standard (Worst-case): $O(n)$ space and $O(\log n)$ query time.



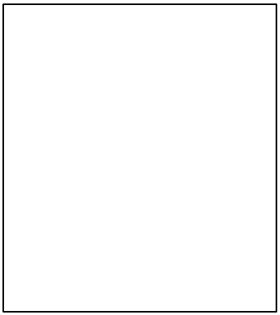
Efficiency

How do we measure efficiency?

- Space: How much storage is needed to store the data structure?
- Query Time: How much time is needed to answer a query?

Let n denote the total space needed to store the map: vertices, edges, faces

Gold Standard (Worst-case): $O(n)$ space and $O(\log n)$ query time.



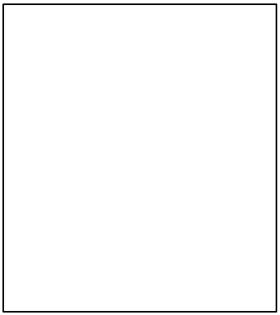
Efficiency

How do we measure efficiency?

- Space: How much storage is needed to store the data structure?
- Query Time: How much time is needed to answer a query?

Let n denote the total space needed to store the map: vertices, edges, faces

Gold Standard (Worst-case): $O(n)$ space and $O(\log n)$ query time.



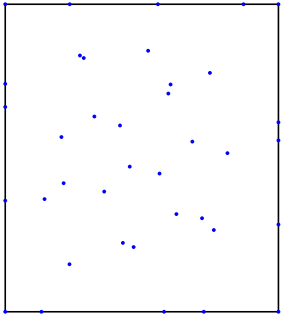
Efficiency

How do we measure efficiency?

- Space: How much storage is needed to store the data structure?
- Query Time: How much time is needed to answer a query?

Let n denote the total space needed to store the map: vertices, edges, faces

Gold Standard (Worst-case): $O(n)$ space and $O(\log n)$ query time.



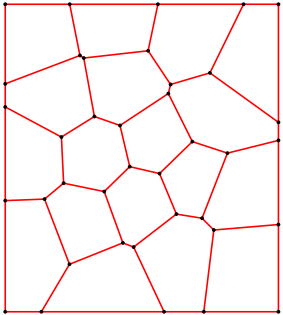
Efficiency

How do we measure efficiency?

- Space: How much storage is needed to store the data structure?
- Query Time: How much time is needed to answer a query?

Let n denote the total space needed to store the map: vertices, edges, faces

Gold Standard (Worst-case): $O(n)$ space and $O(\log n)$ query time.



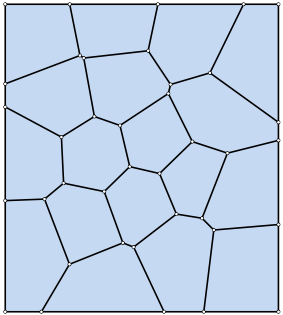
Efficiency

How do we measure efficiency?

- Space: How much storage is needed to store the data structure?
- Query Time: How much time is needed to answer a query?

Let n denote the total space needed to store the map: vertices, edges, faces

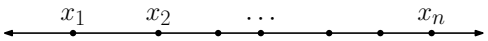
Gold Standard (Worst-case): $O(n)$ space and $O(\log n)$ query time.



1-Dimensional Analog

Point location on a line

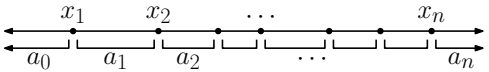
- A set of n points on the line $X = \{x_1, \ldots, x_n\}$
- ... defines $n + 1$ intervals $A = \{a_0, \ldots, a_n\}$, where $a_i = [x_i, x_{i+1}]$
- ... given a query point q, find the interval a_i containing q



1-Dimensional Analog

Point location on a line

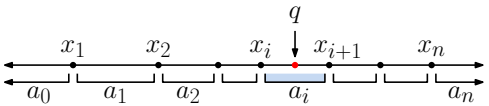
- A set of n points on the line $X = \{x_1, \ldots, x_n\}$
- ... defines $n + 1$ intervals $A = \{a_0, \ldots, a_n\}$, where $a_i = [x_i, x_{i+1}]$
- ... given a query point q, find the interval a_i containing q



1-Dimensional Analog

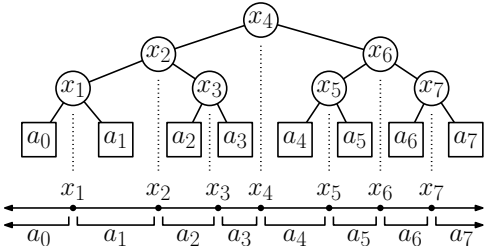
Point location on a line

- A set of n points on the line $X = \{x_1, \ldots, x_n\}$
- ... defines $n + 1$ intervals $A = \{a_0, \ldots, a_n\}$, where $a_i = [x_i, x_{i+1}]$
- ... given a query point q, find the interval a_i containing q



Binary Search

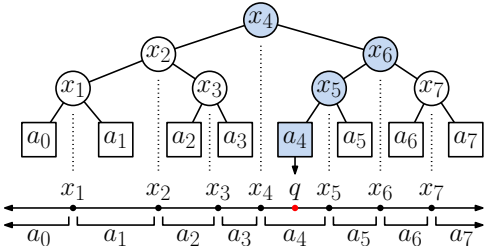
Worst-case optimal: Binary search or generally balanced binary search tree



- $O(n)$ space and $O(\log n)$ query time
- Assuming comparisons, this is optimal. (Note that hashing doesn't apply)

Binary Search

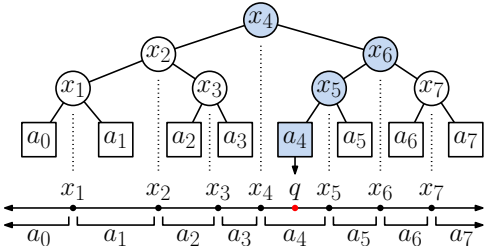
Worst-case optimal: Binary search or generally balanced binary search tree



- $O(n)$ space and $O(\log n)$ query time
- Assuming comparisons, this is optimal. (Note that hashing doesn't apply)

Binary Search

Worst-case optimal: Binary search or generally balanced binary search tree



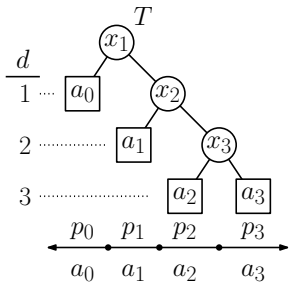
- $O(n)$ space and $O(\log n)$ query time
- Assuming comparisons, this is optimal. (Note that hashing doesn't apply)

Expected-case Complexity

- What if some regions are more likely to be accessed than others? (Manhattan versus the Mohave Desert)
- Each region a_i is associated with an access probability p_i, where $\sum_{i=0}^{n} p_i = 1$.
- Given a binary search tree T, let $d_i(T)$ denote the depth of leaf a_i in T.
- Expected search time:

$$E(T) = \sum_{i=0}^{n} p_i \cdot d_i(T).$$

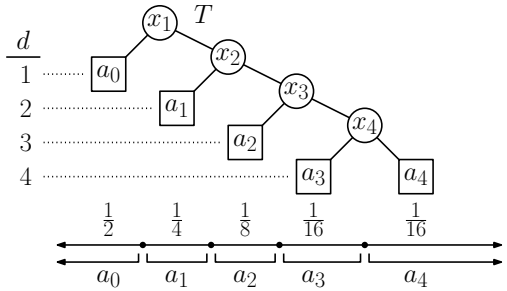
(also called the external path length)



Expected-case: Example

- Let $p_0 = 1/2$, $p_1 = 1/4$, $\ldots$, $p_i = 1/2^{i+1}$.
- Consider the tree T below. Expected search time:

$$E(T) \leq \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{i}{2^i} + \cdots \leq 2 = O(1)$$



Optimal Binary Search Tree

Optimal Binary Search Tree

Given a sequence of points $X = \langle x_1, \ldots, x_n \rangle$, and associated access probabilities $P = \langle p_1, \ldots, p_n \rangle$, construct a tree T that minimizes the expected search time $E(T)$.

What's known:

- $O(n^3)$: Gilbert and Moore (1959)
- $O(n^2)$: Knuth (1971)
- $O(n \log n)$: Hu and Tucker (1971)

These algorithms are either too slow or too complex. Is there a simpler alternative?

Two Greedy Heuristics

Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.

Two Greedy Heuristics

Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

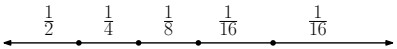
Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.

Heaviest first



Two Greedy Heuristics

Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

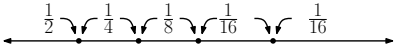
Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.

Heaviest first



Two Greedy Heuristics

Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

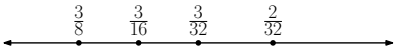
Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.

Heaviest first



Two Greedy Heuristics

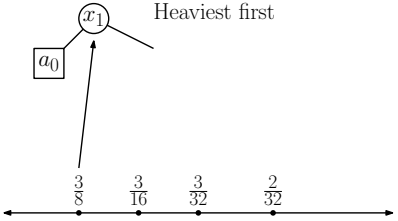
Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.



Two Greedy Heuristics

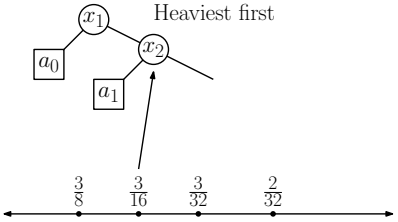
Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.



Two Greedy Heuristics

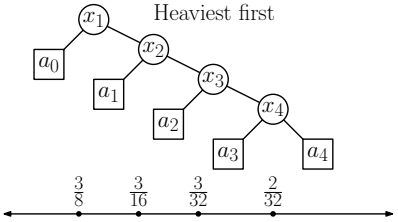
Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.



Two Greedy Heuristics

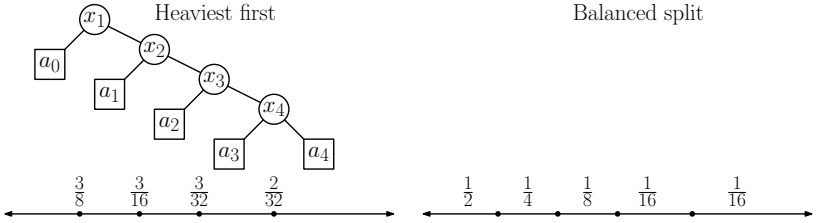
Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.



Two Greedy Heuristics

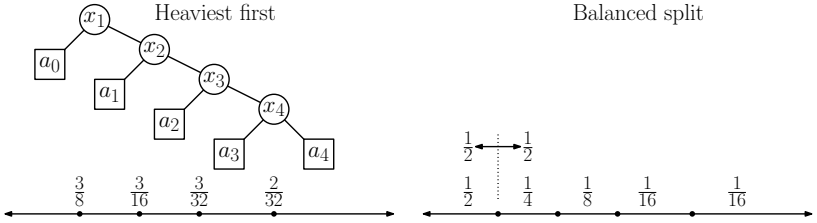
Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.



Two Greedy Heuristics

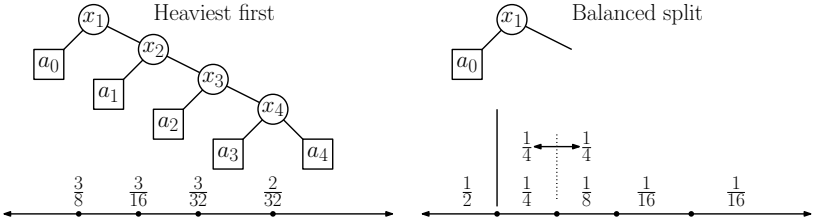
Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.



Two Greedy Heuristics

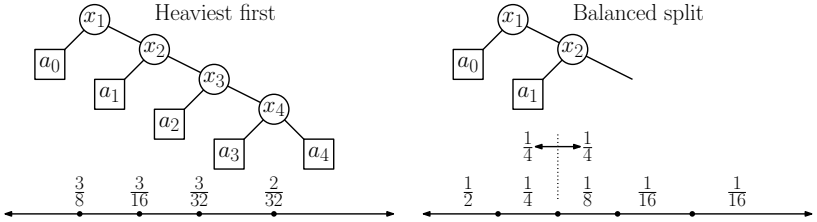
Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.



Two Greedy Heuristics

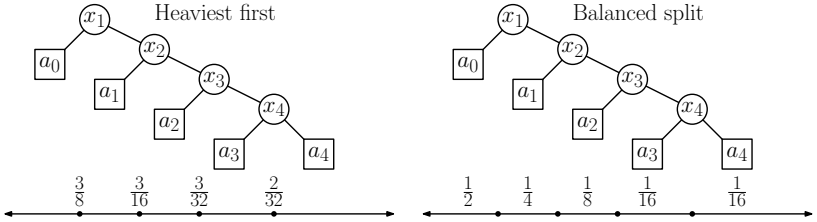
Associate each point x_i with an induced probability, $\tilde{p}_i = (p_{i-1} + p_i)/2$. (Each endpoint steals half the weight of its neighboring intervals.)

Heaviest-first

Split first on x_i that maximizes the induced probability $\tilde{p}_i$. Recur on each side.

Balanced-split

Split first on the x_i that most evenly splits the probability mass. Recur.



Two Greedy Heuristics (Knuth)

In spite of their apparent similarity, one of these heuristics can be arbitrarily bad and one is nearly optimal!

Which is which?

- Heaviest-First: ??
- Balanced-Split: ??

Two Greedy Heuristics (Knuth)

In spite of their apparent similarity, one of these heuristics can be arbitrarily bad and one is nearly optimal!

Which is which?

- Heaviest-First: Arbitrarily bad
- Balanced-Split: Nearly optimal

Why is Heaviest-First So Bad?

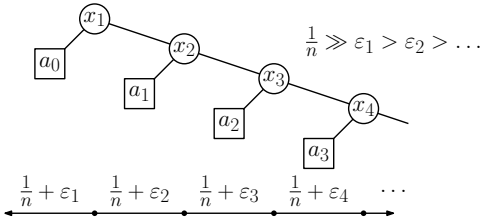
Bad example:

Why is Heaviest-First So Bad?

Bad example: All the probabilities nearly equal, but monotonically decreasing.

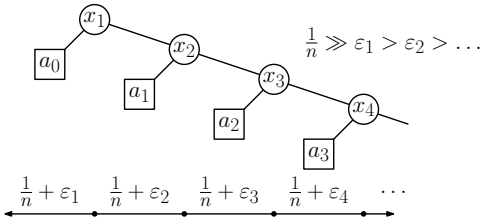
Why is Heaviest-First So Bad?

Bad example: All the probabilities nearly equal, but monotonically decreasing.



Why is Heaviest-First So Bad?

Bad example: All the probabilities nearly equal, but monotonically decreasing.



Optimal tree is balanced. $O(n)$ versus $O(\log n)$.

Is there a way to fix this heuristic? We'll return to this later.

Why is Balanced-Split Fairly Good?

Entropy

Given a discrete probability distribution $P = \langle p_1, \ldots, p_n \rangle$, its entropy is

$$H(P) = \sum_{i=1}^{n} p_i \cdot \log_2 \frac{1}{p_i}$$

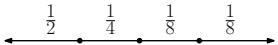
From Shannon's source coding theorem (1948), $H(P)$ is a lower bound on the expected cost of any tree with probabilities $\{p_1, \ldots, p_n\}$, irrespective of their order.

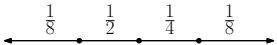
Why is Balanced-Split Fairly Good?

The additional constraint on leaf order further complicates matters.

Why is Balanced-Split Fairly Good?

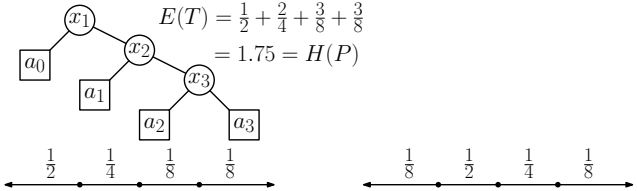
The additional constraint on leaf order further complicates matters.





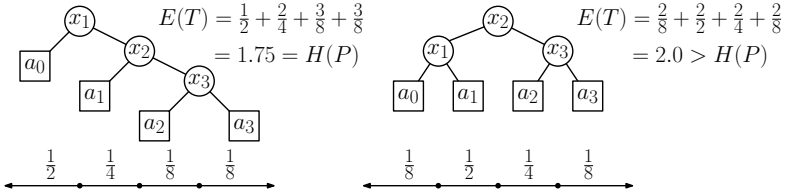
Why is Balanced-Split Fairly Good?

The additional constraint on leaf order further complicates matters.



Why is Balanced-Split Fairly Good?

The additional constraint on leaf order further complicates matters.



Why is Balanced-Split Fairly Good?

Nonetheless, Mehlhorn showed that Balanced-Split is nearly optimal.

Mehlhorn's Analysis (Mehlhorn 1975)

Theorem. Given a probability sequence $P = \langle p_1, \ldots, p_n \rangle$, let T_{opt} be the optimum binary search tree and let T_{bal} be the tree resulting from the Balanced-Split greedy heuristic. Then

$$H(P) \;\leq\; E(T_{\text{opt}}) \;\leq\; E(T_{\text{bal}}) \;\leq\; 2 + 1.44 \cdot H(P)$$

Why is Balanced-Split Fairly Good?

Observation: If the probability mass of each node is at least a constant factor smaller than its parent's, the expected cost will be $O(H(P))$

Key: If one split doesn't make progress, the next will

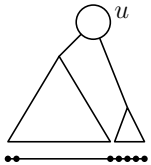
Mehlhorn's Key Lemma

There exists a constant $\delta < 1$ such that the following holds. Consider three internal nodes u, u', and u'' along any path in the balanced split tree, and let w, w', and w'' denote the probability masses of their respective subtrees. Then either: $w' \leq \delta w$ or $w'' \leq \delta^2 w$.

Why is Balanced-Split Fairly Good?

Observation: If the probability mass of each node is at least a constant factor smaller than its parent's, the expected cost will be $O(H(P))$

Key: If one split doesn't make progress, the next will



Mehlhorn's Key Lemma

There exists a constant $\delta < 1$ such that the following holds. Consider three internal nodes u, u', and u'' along any path in the balanced split tree, and let w, w', and w'' denote the probability masses of their respective subtrees. Then either: $w' \leq \delta w$ or $w'' \leq \delta^2 w$.

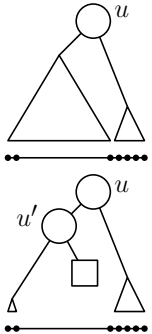
Why is Balanced-Split Fairly Good?

Observation: If the probability mass of each node is at least a constant factor smaller than its parent's, the expected cost will be $O(H(P))$

Key: If one split doesn't make progress, the next will

Mehlhorn's Key Lemma

There exists a constant $\delta < 1$ such that the following holds. Consider three internal nodes u, u', and u'' along any path in the balanced split tree, and let w, w', and w'' denote the probability masses of their respective subtrees. Then either: $w' \leq \delta w$ or $w'' \leq \delta^2 w$.

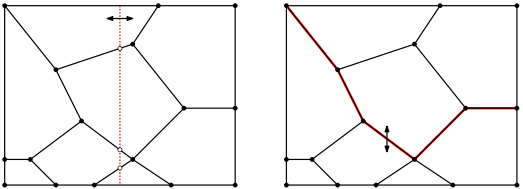


Back to Point Location

Unfortunately, neither heuristic leads to a good solution to point location.

How to split a subdivision?

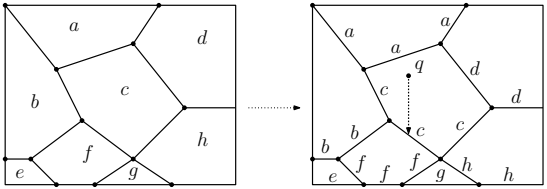
- Split by a line: Results in fragmentation—space may blow up.
- Split by a path: Evaluating a node may be slow—query time may blow up.



Point Location and Vertical Ray Shooting

Reducing Point Location to Vertical Ray Shooting:

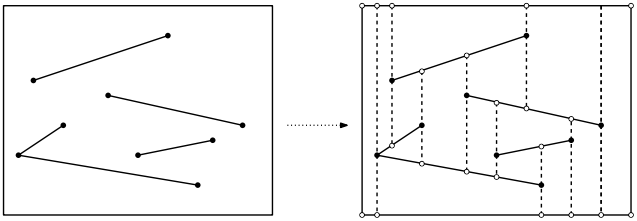
- Label each edge of a subdivision with the region that lies directly above it.
- To determine the region containing q, shoot a vertical ray down an determine the segment it hits.
- The region above this segment is the answer to the query.



Henceforth, we think of the subdivision as a set of (nonvertical) line segments.

Vertical Ray Shooting and Trapezoidal Maps

Trapezoidal Map: Given a set of (nonintersecting, nonvertical) line segments, shoot a bullet path up and down through each endpoint. The result is a subdivision into trapezoids with vertical sides.

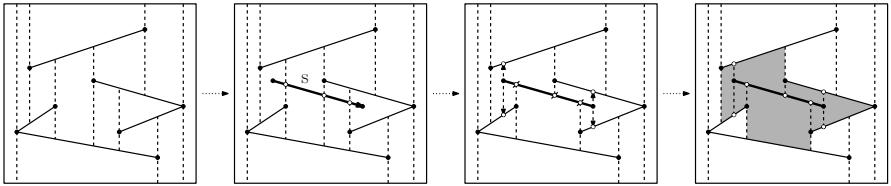


Vertical ray shooting queries can be solved by constructing a trapezoidal map of the segments, and locating the trapezoid containing the query point.

Building a Trapezoidal Map

Building a Trapezoidal Map: Add segments one-by-one, and update the map.

- Find the trapezoid containing the segment's left endpoint. Shoot its bullets.
- "Walk" the segment through the map, discovering vertical walls that are hit
- Trim the bullet paths that hit the segment



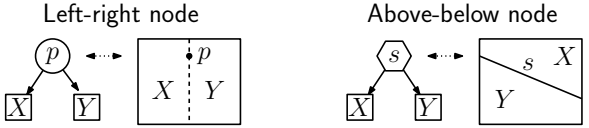
The total space is $O(n)$. If segments are inserted in random order, then the expected running time is $O(n \log n)$.

History DAG

Cool fact: You can build an efficient point-location data structure while you are building the map.

We will need two types of discriminators (node types):

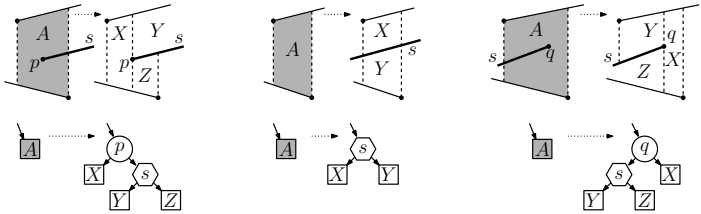
- Left-right node: Given a point, this node splits space by a vertical line through this point
- Above-below node: Given a (nonvertical) line segment, this node splits space above and below this line



History DAG

Building the Point Location Structure:

- Maintain a tree (actually a rooted DAG) whose leaves are the trapezoids.
- As new trapezoids replace old ones, add internal nodes to discriminate among the new trapezoids



This DAG effectively stores the history of the construction process—History DAG.
If segments inserted in random order, the expected worst-case query time is $O(\log n)$.

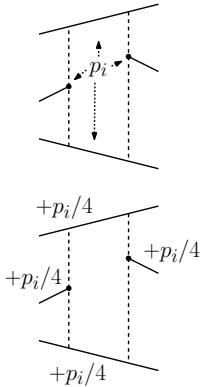
Fast Expected-Case Point Location

Let $P = \{p_1, \ldots, p_n\}$ be the probability distribution of the trapezoids

Objective: Build a search structure whose query time is $O(H(P))$

Approach: (Slightly simplified)

- Each trapezoid of the map is bounded by at most four segments (top, bottom, left, right)
- Induced probability: Split the probability p_i of each trapezoid into quarters, and give these to the four incident segments
- Weighted randomized insertion: Insert the segments in weighted random order and output the resulting the history graph



Fast Expected-Case Point Location

Main Result

Consider a trapezoidal map of n segments with access probabilities P. The weighted randomized incremental algorithm produces a point location structure of size $O(n)$ that answers queries in average time $(5 \ln 2)H(P) + O(1)$.

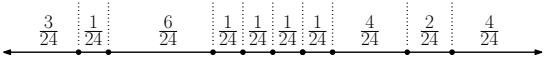
Intuition:

- The analysis is based on assigning a number of pebbles to the segments proportional to the induced probability
- Sample pebbles at random. The first sampled pebble associated with a segment inserts that segment
- A segment s_i with induced probability $\tilde{p}_i$ is inserted at level $O(\log 1/\tilde{p}_i)$ in expectation
- All the segments bounding a trapezoid of probability p_i are inserted at level $O(\log 1/p_i)$ in expectation

Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

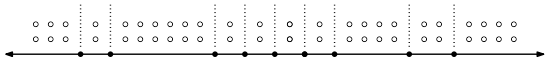
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

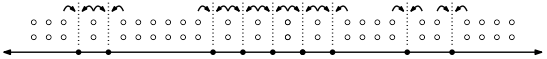
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

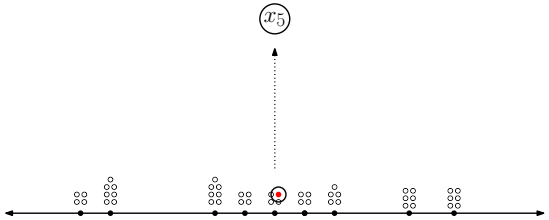
Simplified Justification: (Based on the 1-dimensional case)

- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$

Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

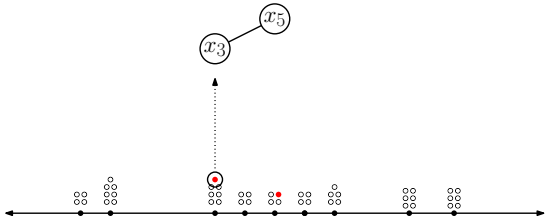
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

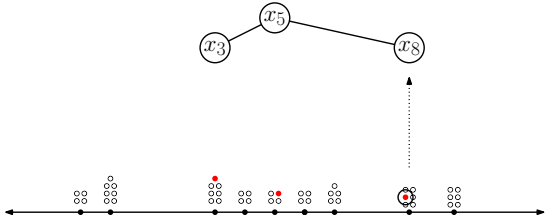
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

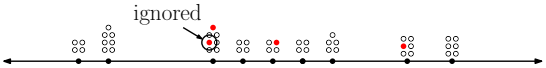
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

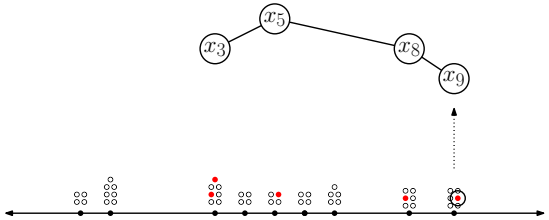
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

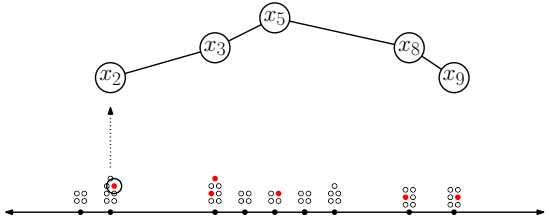
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

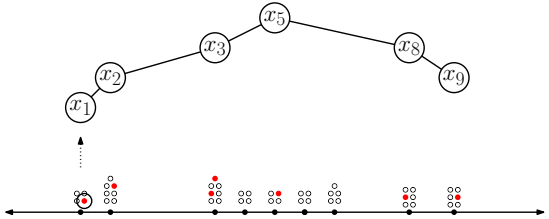
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

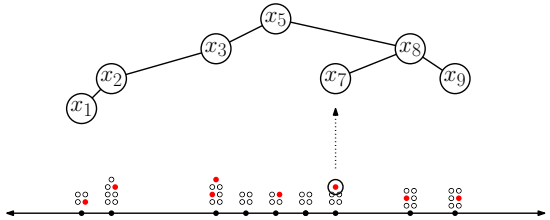
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

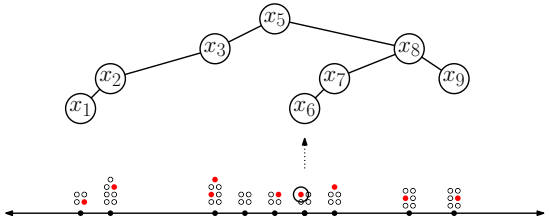
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

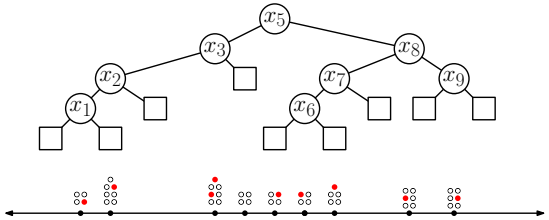
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

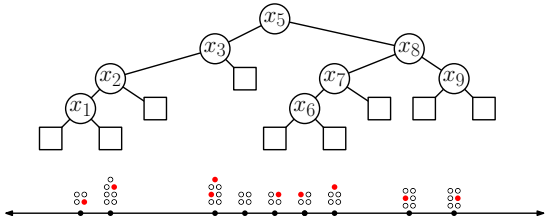
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

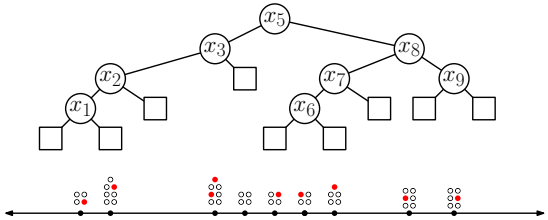
- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Example: 1-Dimensional Analog

Simplified Justification: (Based on the 1-dimensional case)

- Assign interval of probability p_i, $2\lceil np_i \rceil$ pebbles
- Redistribute half of each interval's pebbles to its two endpoints
- Sample pebbles at random. The first pebble for x_i inserts x_i
- Endpoint x_i is inserted by after $O(1/\tilde{p}_i)$ pebbles in expectation
- Random insertions lead to a balanced tree, so expected depth is $O(\log 1/\tilde{p}_i)$



Concluding Remarks

- Heuristics (heaviest-first and balanced-split) for building binary search trees for good expected-case search time
- Showed that balanced-split heuristic is nearly optimal, $2 + O(H(P))$.
- Presented a randomized incremental construction of trapezoidal maps as an approach to point location
- A weighted version of the randomized construction provides good expected-case point location
- Although heaviest-first was shown to be bad, the above result implies that randomized heaviest-first is actually good.
- Open: Dynamic updates?

Merci!

Bibliography

- E. N. Gilbert and E. F. Moore, Variable-Length Binary Encodings, *Bell System Tech. J.* 38, 1959, 933–967.
- T. C. Hu and A. C. Tucker, Optimal Computer Search Trees and Variable-Length Alphabetical Codes, *SIAM J. Appl. Math.* 21, 1971.
- D. E. Knuth, Optimum Binary Search Trees, *Acta Inform.* 1, 1971, 14–25.
- K. Mehlhorn, Nearly Optimal Binary Search Trees, *Acta Inform.* 5, 1975, 287–295.
- C. E. Shannon, A Mathematical Theory of Communication, *Bell System Technical Journal*, 27, 379–423, 623–656, 1948.