Orthogonal Range Searching

Benjamin Sach

Orthogonal range searching

A $d$-dimensional range searching data structure stores $n$ distinct points
each point has $d$ coordinates

(we assume $d$ is a constant)

for $d=1$, the lookup$(x_1, x_2)$ operation
returns every point with $x_1 \leq x \leq x_2$.

for $d=2$, the lookup$(x_1, x_2, y_1, y_2)$ operation
returns every point with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$.

for $d=3$, the lookup$(x_1, x_2, y_1, y_2, z_1, z_2)$ operation
returns every point with $x_1 \leq x \leq x_2$, $y_1 \leq y \leq y_2$, and $z_1 \leq z \leq z_2$.

Starting simple... 1D range searching

Alternatively we could build a balanced tree...

The tree has $O(n \log n)$ depth and can be built in $O(n \log n)$ time.

We can store the tree in $O(n)$ space (it has one node per point).

(If the points are sorted)

Starting simple... 1D range searching

How do we do a lookup?

look at any node on the path

"it's all or nothing"

Step 1: Find the successor of $x_1$ in $O(\log n)$ time

Step 2: Find the predecessor of $x_2$ in $O(\log n)$ time

which points in the tree should we output?

those in the $O(\log n)$ selected subtrees on the path
Warning: the root to split path isn’t to scale

1D range searching summary

- lookup \((x_1, x_2)\) should report all points between \(x_1\) and \(x_2\)
- preprocess \(n\) points on a line
- \(O(n \log n)\) prep time
- \(O(n)\) space
- \(O(\log n + k)\) lookup time

where \(k\) is the number of points reported

(this is known as being output sensitive)

2D range searching

A 2D range searching data structure stores \(n\) distinct \((x, y)\) pairs and supports:

- the \(\text{lookup}(x_1, x_2; y_1, y_2)\) operation

which returns every point in the rectangle \([x_1 : x_2] \times [y_1 : y_2]\)

i.e. every \((x, y)\) with \(x_1 \leq x \leq x_2\) and \(y_1 \leq y \leq y_2\).

Attempt one:

- Find all the points with \(x_1 \leq x \leq x_2\)
- Find all the points with \(y_1 \leq y \leq y_2\)
- Find all the points in both lists

How long does this take?

\[O(\log n + k_x) + O(\log n + k_y) + O(k_x + k_y)\]

\[= O(\log n + k_x + k_y)\]

these could be huge in comparison with \(k\)

here \(k_x\) is the number of points with \(x_1 \leq x \leq x_2\)

respectively for \(k_y\)
**Subtree decomposition in 2D**

**Query summary**
1. Follow the paths to $x_1$ and $x_2$ (inspecting the points on the path as you go)
2. Discard off-path subtrees where the $x$ coordinates are too large or too small
3. For each off-path subtree where the $x$ coordinates are in range...
   * use the 1D range structure for that subtree to filter the $y$ coordinates

**How long does a query take?**
- The paths have length $O(\log n)$
- So steps 1. and 2. take $O(\log n)$ time
- As for step 3, we do $O(\log n)$ 1D lookups...
  - Each takes $O(\log n + k')$ time
  - This sums to $O(\log^2 n + k)$ because the 1D lookups are disjoint

**Space Usage**
- How much space does our 2D range structure use?
  - the original (1D) structure used $O(n)$ space...
  - but we added some stuff
  - at each node we store an array containing the points in its subtree
  - the array is sorted by $y$ coordinate (this gives us a 1D range data structure)
- Look at any level in the tree i.e. all nodes at the same distance from the root
- the points in these subtrees are disjoint
- so the sizes of the arrays add up to $n$
- As the tree has depth $O(\log n)$...
  - the total space used is $O(n \log n)$

**Preprocessing time**
- How much prep time does our 2D range structure take?
  - the original (1D) structure used $O(n \log n)$ prep time...
  - but we added some stuff
  - How long does it take to build the arrays at the nodes?
- $\ell$ is just merged with
- as and are already sorted, merging them takes $O(\ell)$ time
- Therefore the total time is $O(n \log n)$ (which is the sum of the lengths of the arrays)

**2D range searching**
- A 2D range searching data structure stores $n$ distinct $(x,y)$-pairs and supports:
  - the lookup($x_1,x_2,y_1,y_2$) operation which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$
  - i.e. every $(x,y)$ with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$

**Summary**
- $O(n \log n)$ prep time
- $O(n \log n)$ space
- $O(\log^2 n + k)$ lookup time
  - where $k$ is the number of points reported
  - actually we can improve this :)
The arrays of points at the children partition the array of the parent.

Observation: if we know where the successor of $y_1$ is in the parent, can find the successor in either child in $O(1)$ time.

Adding these links doesn’t increase the space or the prep time.

2D range searching

A 2D range searching data structure stores $n$ distinct $(x, y)$ pairs and supports:

- The lookup $(x_1, x_2, y_1, y_2)$ operation which returns every point in the rectangle $[x_1 : x_2] \times [y_1 : y_2]$

i.e. every $(x, y)$ with $x_1 \leq x \leq x_2$ and $y_1 \leq y \leq y_2$.

Summary

- $O(n \log n)$ prep time
- $O(n \log n)$ space
- $O(\log n \times k)$ lookup time

where $k$ is the number of points reported.

The improved query time

How long does a query take?

The paths have length $O(\log n)$.

So steps 1. and 2. take $O(\log n)$ time.

As for step 3, we do $O(\log n \times \log n)$ 1D lookups.

Each takes $O(k')$ time.

This sums to:

$O(\log n \times k)$

Query summary

1. Follow the paths to $x_1$ and $x_2$ (updating the successor to $y_1$ as you go).
2. Discard off-path subtrees where the $x$ coordinates are too large or too small.
3. For each off-path subtree where the $x$ coordinates are in range, use the 1D range structure for that subtree to filter the $y$ coordinates.

We improved this :) using fractional cascading.