kd-Trees Continued

Generalized, incremental NN, range searching, kd-tree variants
kd-tree Variants

- How do you pick the cutting *dimension*?
  - kd-trees cycle through them, but may be better to pick a different dimension
  - e.g. Suppose your 3d-data points all have same Z-coordinate in a given region:

- How do you pick the cutting *value*?
  - kd-trees pick a key value to be the cutting value, based on the order of insertion
  - *optimal kd-trees*: pick the key-value as the median
  - Don’t need to use key values => like PR Quadtrees => PR kd-trees

- What is the size of leaves?
  - if you allow more than 1 key in a cell: *bucket kd-trees*

- kd-trees: discriminator = (hyper)plane; quadtrees (and higher dim) discriminator complexity grows with $d$
Sliding Midpoint kd-trees

- PR kd-tree: split in the midpoint, along the current cutting dimension
- May result in trivial splits: if all points lie to one side of the median
- Solution: if you get a trivial split, slide the split so that it cuts off at least one point:

Avoids empty cells

Tends to put boundaries around bounding boxes of clusters of points
**kd-Trees vs. Quadtrees, another view**

Consider a 3-d data set

![Octtree Diagram]

Octtree

![kd-tree Diagram]

kd-tree splits the decision up over d levels
don’t have to represent levels (pointers) that you don’t need

**Quadtrees**: one *point* determines all splits

**kd-trees**: flexibility in how splits are chosen
Path-compressed PR kd-trees

Strings of Ls and Rs tell the decisions skipped that would lead to this node

Path compressed PR kd-tree
Generalized Nearest Neighbor Search

- Saw last time: nearest neighbor search in kd-trees.
- What if you want the k-nearest neighbors?
- What if you don’t know k?
  - E.g.: Find me the closest gas station with price < $3.25 / gallon.
  - Approach: go through points (gas stations) in order of distance from me until I find one that meets the $ criteria
- Need a NN search that will find points in order of their distance from a query point \( q \).
- Same idea as the kd-tree NN search, just more general
Generalized NN Search

• A feature of all spatial DS we’ve seen so far: decompose space hierarchically.
  No matter what the DS, we get something like this:

Let the items in the hierarchy be e1,e2,e3...

Items may represent points, or bounding boxes, or ...

Let Type(e) be an abstract “type” of the object:
  we use the type to determine which distance function to use
  E.g: if Type = “bounding box” then we’d use the point-to-rectangle distance function.

A concrete example: in a Quadtree: internal nodes have type “bounding box”
Leaves would have type “point”
**Generalized, Incremental NN**

Let `IsLeaf()`, `Children()`, and `Type()` represent the decomposition tree

Let $d_t(q, e_t)$ be the distance function appropriate to compare points with elements of type $t$.

Idea: keep a priority queue that contains all elements visited so far (points, bounding boxes)

Priority queue (heap) is ordered by distance to the query point

When you dequeue a point (leaf), it will be the next closest

```python
HeapInsert(H, root, 0)
while not Empty(H):
    e := ExtractMin(H)
    if IsLeaf(e):
        output e as next nearest
    else
        foreach c in Children(e):
            t = Type(c)
            HeapInsert(H, c, $d_t(q,c)$)
```

$d_t(q,c)$ may be the distance to the bounding box represented by $c$, e.g.
Incremental, Generalized NN Example

HeapInsert(H, root, 0)

while not Empty(H):
    e := ExtractMin(H)
    if IsLeaf(e):
        output e as next nearest
    else
        foreach c in Children(e):
            t = Type(c)
            HeapInsert(H, c, d_t(q,c))
Incremental, Generalized NN Example

Some spatial data structure:

```
HeapInsert(H, root, 0)
while not Empty(H):
    e := ExtractMin(H)
    if IsLeaf(e) && IsPoint(e):
        output e as next nearest
    else
        foreach c in Children(e):
            t = Type(c)
            HeapInsert(H, c, d_t(q,c))
```

Its spatial decomposition (NOT the actual data structure):

```
H = []
H = [T]
H = [L_T R_T]
H = [A_Q R_T B_Q]
H = [R_T B_Q]
H = [B_S A_S B_Q]
H = [As a B_Q]
H = [c a B_Q]
H = [c a b]
H = [a b]
H = [b]
H = []
```
Range Searching
CMSC 420
Range Searching in kd-trees

• Range Searches: another extremely common type of query.

• Orthogonal range queries:
  - Given axis-aligned rectangle
  - Return (or count) all the points inside it

• Example: find all people between 20 and 30 years old who are between 5’8” and 6’ tall.
Range Searching in kd-trees

- Basic algorithmic idea:
  - traverse the whole tree, **BUT**
    - prune if bounding box doesn’t intersect with Query
    - stop recursing or print all points in subtree if bounding box is entirely inside Query
If query box doesn’t overlap bounding box, stop recursion
If bounding box is a subset of query box, report all the points in current subtree
If bounding box overlaps query box, recurse left and right.
def RangeQueryCount(Q, T):
    if T == NULL: return 0
    if BB(T) doesn’t overlap Query: return 0
    if Query subset of BB(T): return T.size

    count = 0
    if T.data in Query: count++

    count += RangeQuery(Q, t.left)
    count += RangeQuery(Q, t.right)

    return count

(For clarity, omitting the cutting dimension, and the BB(T) parameters that would be passed into the function)
def RangeQuery(Q, T):
    if T == NULL: return empty_set()
    if BB(T) doesn’t overlap Query: return 0
    if Query subset of BB(T): return AllNodesUnder(T)

    set = empty_set()
    if T.data in Query: set.union({T.data})

    set.union(RangeQuery(Q, T.left))
    set.union(RangeQuery(Q, T.right))

    return set
Expected # of Nodes to Visit

- Completely process a node only if query box intersects bounding box of the node’s cell:
- In other words, one of the edges of Q must cut through the cell.
- # of cells a vertical line will pass through ≥ the number of cells cut by the left edge of Q.
- Top, bottom, right edges are the same, so bounding # of cells cut by a vertical line is sufficient.
# of Stabbed Nodes = $O(\sqrt{n})$

Consider a node $a$ with cutting dimension = $x$

Vertical line can intersect exactly one of $a$’s children (say $c$)

But will intersect both of $c$’s children.

Thus, line will intersect at most 2 of $a$’s grandchildren.
# of Stabbed Nodes = O(√n)

So: you at most double # of cut nodes every 2 levels

If kd-tree is balanced, has O(log n) levels

Cells cut
\[= 2^{(\log n)/2}\]
\[= 2^{\log \sqrt{n}}\]
\[= \sqrt{n}\]

Assuming random input, or all points known ahead of time, you’ll get a balanced tree.

Each side of query rectangle stabs < O(√n) cells. So whole query stabs at most O(4√n) = O(√n) cells.
Suppose we want to output all points in region

- Then cost is $O(k + \sqrt{n})$
  - where $k$ is # of points in the query region.
- Why? Because: you visit every stabbed node [$O(\sqrt{n})$ of them] +
  every node in the subtrees rooted in the contained cells.
  - Takes linear time to traverse such subtrees
- Example of output sensitive running time analysis: running time depends on size of the output.

\[
\text{AllNodesUnder}(a)
\]
kd-tree Summary:

• Use $O(n)$ storage [1 node for each point]

• If all points are known in advance, balanced kd-tree can be built in $O(n \log n)$ time
  - Recall: sort the points by x and y coordinates
  - Always split on the median point so each split divides remaining points nearly in half.
  - Time dominated by the initial sorting.

• Can be orthogonal range searched in $O(\sqrt{n} + k)$ time.

• Can we do better than $O(\sqrt{n})$ to range search?
  - (possibly at a cost of additional space)