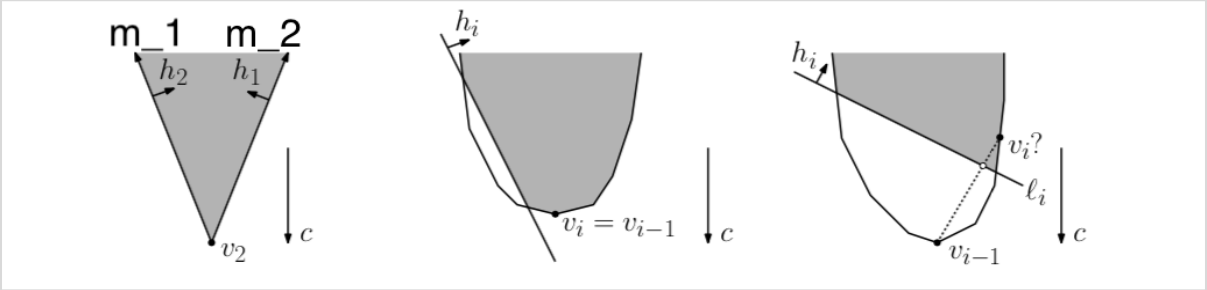
CS 6301.008.18S Lecture—January 30, 2017

Main topics are #linear_programming, #backwards_analysis, #range_searching, and #kd-trees.

Linear Programming Continued

- Last, we looked at an algorithm for linear programming. We're given halfplanes {h_1, …, h_n} described as linear constraints and an objective vector c.
- Let M be some really really big number. So big that the optimal vertex has coordinates less than M. We'll add two constraints
  - m_1 := {x_1 ≤ M if c_1 > 0, -x_1 ≤ M otherwise}
  - m_2 := {x_2 ≤ M if c_2 > 0, -x_2 ≤ M otherwise}
- Our initial optimal solution will lie at the intersection of m_1 and m_2.



- Let H_i := {m_1, m_2, h_1, h_2, …, h_i} be our two new halfplanes and the first i original halfplanes. Let ell_i be the line bounding h_i. Let C_i := m_1 ∩ m_2 ∩ h_1 ∩ h_2 ∩ … ∩ h_i be their intersection. Let v_i be the optimal vertex in C_i.
- 2DBoundedLP(H = {h_1, …, h_n}, c, m_1, m_2):
  - v_0 ← corner of C_0
  - for i ← 1 to n
    - if v_{i-1} in h_i
      - v_i ← v_{i-1}
    - else
      - p ← the point on ell_i maximizing objective subject to H_{i-1}
      - if p does not exist
        - return "Infeasible!"
      - else
        - v_i ← p
  - return v_n
- This algorithm takes O(n^2) time.
- But you can do better by randomizing the order of the halfplanes.
- We can pick a permutation uniformly at random in only O(n) time.
- Let X_i be a random variable where X_i = 1 if v_{i-1} not in h_i and X_i = 0 otherwise.

- The time spent solving 1D linear programs to move the optimal vertex is itself a random variable equal to O(n) + sum_{i = 1}^n O(i) * X_i.
- By linearity of expectations, E[O(n) + sum_{i=1}^n O(i) * X_i] = O(n) + sum_{i=1}^n O(i) * E[X_i].
- Each E[X_i] is equal to the probability that v_{i-1} not in h_i.
- But how do we compute that? It may be tempting to look at C_{i-1} and then ask what is the probability any of the remaining n - i + 1 halfplanes change the optimal vertex. But now we're conditioning on what C_{i-1} looks like and things are bit ugly.
- Instead, we'll use something called *backwards analysis* where we look at C_i and just ask about what happens before and up to its creation.
- Consider v_i. It lies at the intersection of two halfplane boundaries. If *neither* boundary was h_i, then v_i was already the lowest feasible vertex before adding h_i, and v_{i-1} = v_i.
- Now, if you fix v_i and the set of halfplanes h_1 through h_i that led to it, *but not their order* then all permutations of that subset h_1 through h_i are equally likely, and all choices for h_i out of that subset are equally likely.
- In particular, one of those two halfplanes is chosen with probability at most 2/i. (The probability is actually less if one of the two halfplanes was m_1 or m_2, since neither can be the last halfplane chosen.)
- This upper bound on the probability is true no matter which subset of halfplanes makes up h_1 through h_i, so E[X_i] ≤ 2/i.
- Therefore, O(n) + sum_{i=1}^n O(i) * E[X_i] ≤ O(n) + sum_{i = 1}^n O(i) * 2 / i = O(n).
- The expected running time is linear!

Higher Dimensions and Closeness to Expectation

- So what if d > 2?
- At a high level, the algorithm changes very little. That step where we find a point on a line turns into finding a point on a hyperplane of dimension d -1. To do that, we call the same algorithm recursively but in one lower dimension. The base case is d = 1, which is just that line searching procedure.
- The run time analysis is similar but a fair bit more messy. You can look at the book or notes if you're curious. The running time comes out to be O(d! n). So for constant d, the algorithm still runs in linear time. But d! grows so fast that it's really only practical for very small values of d.
- Now, you may be worried that we only analyzed the expected running time. Is there some small probability that the algorithm is still slow?
- Yes, but it's extremely unlikely. The probability you exceed the expected running time by a factor of b is O((1/c)^{bd!}) for any fixed constant c. This gets very very small as d or b increase.

- Even in 2D, the probably you exceed the expectation by a factor of 10 is less than the probability of getting hit by lightening *twice* in your lifetime.

Range Searching

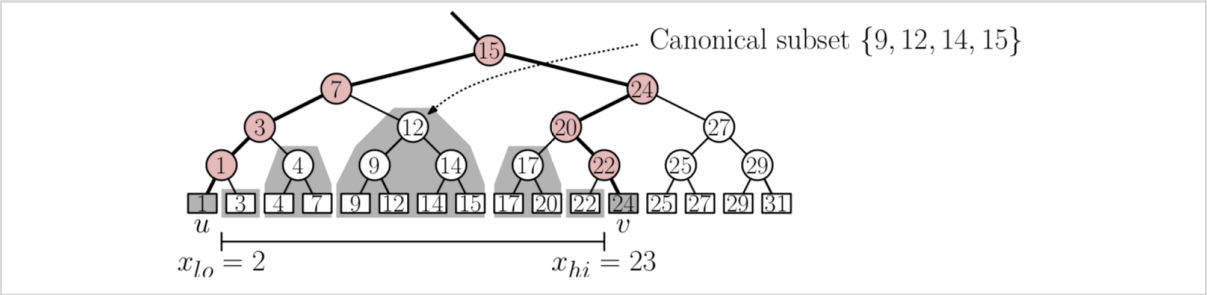
- For the next couple lectures, we're going to consider a different kind of problem.
- Let's say you are given a set of n points P and a class of *range shapes* like rectangles, balls, etc.. You want to build a data structure to help speed up certain operations.
- Specifically, we'll later be given one or more query ranges Q, and we want to use our data structure to learn about the points of P lying in Q quickly. Again, we know what kind of shape Q is in advance, but not which specific Q we care about.
- There's a few ways to define learning about P. For example…
- *Range reporting* is returning a list of all points of P lying in Q.
- *Range counting* is returning a *count* of the points of P lying in Q. You could also give each point p a weight w(p) and return the sum of the weights in Q.
- There are lots of different data structures depending on what kinds of shapes Q can be. We're going to focus on orthogonal rectangular range queries, where we're guaranteed Q is an axis-parallel rectangle.
- Imagine each point is a person in a database and each coordinate tells you some statistic like their age, salary, etc. These queries are asking for all people in a certain age range, with a certain salary range, etc.
- Most data structures for range searching of any type rely on an idea called canonical subsets.
- A collection of *canonical subsets* {P_1, …, P_k} with each P_i in P is chosen so that any intersection of P and an allowable range shape can be formed from the *disjoint* union of canonical subsets. The subsets from the list may overlap, but the subsets for a particular range query do not.
- The hard part is finding a small collection of canonical subsets so you don't waste space, and finding a way to quickly pick the right ones during a query so you don't waste time.
- Typically, we define canonical subsets using a *partition tree*. This is a rooted, usually binary, tree whose leaves are points of P. Each node u is associated with some subset of P, in particular the points stored at the leaves of u's subtree.

One-dimensional Range Queries

- Let's start with the simple example of orthogonal rectangular range queries in 1D. These are simply *interval queries*.
- So, P = {p_1, p_2, …, p_n} and each query is an interval [x_lo, x_hi].
- Here's a data structure we can use: sort the points of P from left to right and store them in

the leaves of a balanced binary search tree.

- Each internal node is labeled with the largest x-coordinate of a descendent leaf.
- Each internal node is also associated with the canonical subset of P equal to points in its descendent leaves.

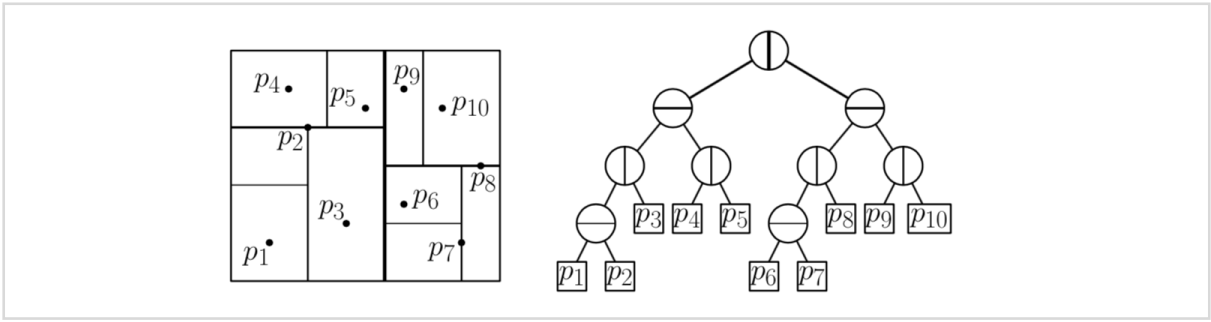


- But! We don't store the canonical subsets explicitly. If you want to report the points in a node's canonical subset, you only need to traverse its subtree. If you want to do quick range counting, you instead record on each node the number/total weight of its points.
- There are O(n) leaves, so it uses O(n) space total. (We can also build it bottom up in O(n log n) time.)
- So how do we do a query with this data structure?
- Find the rightmost leaf u with key less than x_lo and find the leftmost leaf v with key greater than x_hi. The leaves strictly between u and v are the points in the range.
- To make counting fast, it's better if we find a small collection of canonical subsets that contain all the query points. We'll take those maximal rooted subtrees shaded in grey.
- So, follow the paths to u and v from the root until they diverge. After divergence, if stepping left toward u from w, take the canonical subset of w's right child. Similarly, if stepping right toward v from w, take the canonical subset of w's left child.
- To count, sum over the total counts for all those canonical subsets in constant time per subset. For reporting, look at the leaves in those subtrees in time linear in the size of each subset.
- We touch O(log n) canonical subsets during a query. Counting takes O(log n) time per query. Reporting takes O(log n + k) time where k is the number of points reported.

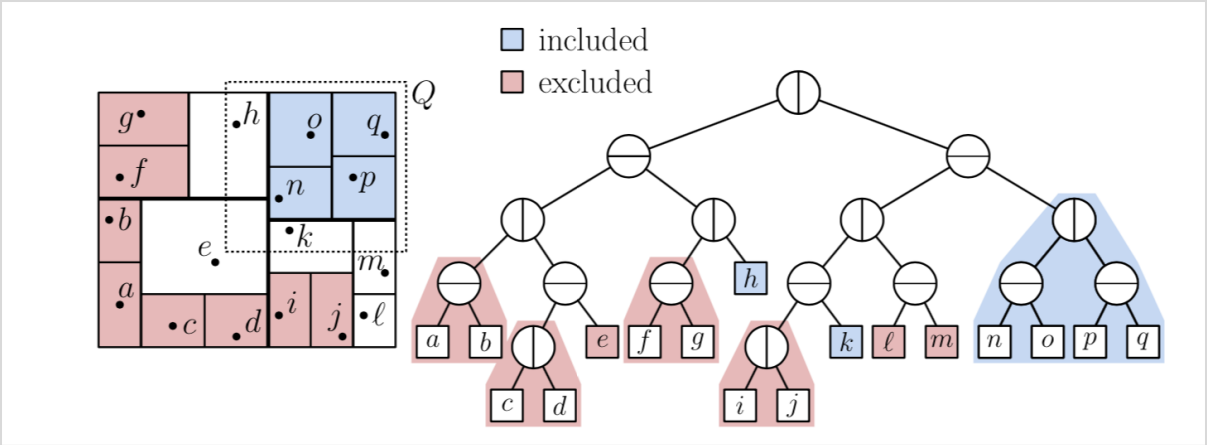
kd-trees

- So what about the plane or even higher dimensions?
- We'll start with a data structure called the kd-tree, designed by [Bentley '75].
- So originally, this stood for k-dimensional tree. But using k for dimension is confusing, and people forgot that's how it worked. So now we'll say things like 2 or 3-dimensional kd-tree.
- kd-trees are partition trees. At each node, we subdivide its point set by splitting them evenly based on their x-coordinate or y-coordinate.
- Each node t stores
  - t.cut-dim: which way we'll split the points (so 0 for x and 1 for y)

    - t.cut-val: at which x or y coordinate should we split the points
    - t.weight: the total weight or number of points in t's subtree
- So if t.cut-dim is 0 and t.cut-val is 400, we'll store points of x-value < 400 in the left subtree and points of higher x-value in the right subtree. We'll break ties so that the two subtrees are as evenly split as possible.
- Nodes with one point are the leaves. They store that point as t.point.



- If you zoom out a bit, here is the picture you see. Each node represents a rectangular region of space called a *cell*. The root's cell can be thought of as a big rectangle surrounding all the points. When you split a node's points, it's like you're splitting its cell into two smaller cells for that node's children.
- These nested cells are sometimes called a *hierarchical space decomposition*.
- Now, there's a few ways you can pick cut dimension and cut value. The standard way is to alternate between x and y as the cut dimension as I drew, and always set cut-val to be the median value so you split the points into two equal subsets.
- You can build this thing in O(n log n) time by first making two sorted lists of points by x-coordinate and y-coordinate. Then you can search for the median coordinate for each split and split up the lists to recursively build the two subtrees in time linear in the number of points in a subtree. You get a recurrence like T(n) = 2T(n/2) + n which solves to O(n log n).
- It's a balances binary tree with O(n) leaves, so its size is O(n).
- We can do range counting with the following procedure:
- RangeCount(Q: the range, u: a node):
    - if u is a leaf
        - if u.point in Q, return u.weight
        - else return 0
    - else
        - if u.cell intersect Q = emptiest, return 0
        - else if u.cell subset Q, return u.weight
        - else, return RangeCount(Q, u.left) + RangeCount(Q, u.right)
- Basically, return the whole count if the whole cell lies in the range. Or return 0 if the whole cell lies outside the range. Otherwise, search deeper.



- If you want to report, then instead of returning the weight, you return a list of all the node's descendent leaves.
- How long does a query take? Call a node *expanded* if it and both children are visited by the recursive counting algorithm. Except for the root, each visited node has an expanded parent, so the running time is proportional to the number of expanded nodes.
- A cell is *stabbed* if it has a strict intersection with the range. There are more stabbed cells than expanded cells, so we'll just count those.
- Lemma: Any horizontal or vertical line stabs O(sqrt(n)) cells of the tree.
- Proof:
  - Suppose the line is vertical. Consider a node with cutting dimension x. The line stabs at most one of its children, and if it fails to stab a child, then it won't stab any descendent of that child.
  - However, a node with cutting dimension y may have both children stabbed.
  - Therefore, each node with cutting dimension x has at most two *grandchildren* stabbed. In general, the number of stabbed nodes increases by a factor of at most two every two levels of the tree.
  - Let S(n) be the maximum number of stabbed nodes by the vertical line. S(n) = 2 if $n \leq 4$ and S(n) = 1 + 2T(n/4) otherwise.
  - This solves to S(n) = O(2^{log_4 n}) = O(n^(1/2)) = O(sqrt(n)).
- OK, for query range Q to stab a cell, that cell must be stabbed by at least one of the four line segments bounding Q. But each of those stabs O(sqrt(n)) cells, so O(sqrt(n)) cells are stabbed or expanded. A counting query takes O(sqrt(n)) time. A reporting query takes O(sqrt(n) + k) time.
- What about higher dimensions? For those, the kd-tree splits along each of the d dimensions, the 0th, the 1st, the 2nd, and so on then back to repeat the list. It still uses O(n) space and can be built in O(n log n) time if d is a constant.
- However, the running time of a counting query increases to O(n^{1-1/d}).
- Next, we'll learn about a data structure that uses slightly more space, but has much better worst-case query time.