CMSC 754: Lecture 14
Orthogonal Range Searching and kd-Trees

Reading: Chapter 5 in the 4M's.

Range Searching: In this lecture we will discuss a new data structure problem,, called *range searching*. We are given a set of n points $P = \{p_1, \ldots, p_n\} \subset \mathbb{R}^d$ and a class of range shapes (e.g., rectangles, balls, triangles, halfplanes). The points of P are to be preprocessed and stored in a data structure (whose structure may be based on the knowledge of the allowable range shapes). Given a query range Q from this class, the objective is identify the points of P that lie within Q. The term "identify" can mean a number of things:

Emptiness: If $P \cap Q = \emptyset$, return "empty" else "non-empty"

Counting: Return the count $|P \cap Q|$ of the number of points of P within Q

Weighted counting: Each point $p_i \in P$ has an associated weight $w(p_i)$, and we are to return the weighted sum of points in Q, $\sum_{p \in Q} w(p_i)$.

Semigroup weighted counting: Why limit to addition? For example, how about computing the product of weights or the maximum of the weights?

Generally, weighted counting makes sense as long as you have an operator that is both commutative and associative. Your data structure may also take advantage of special properties of your operator. For example, if the operator has an *inverse*, you have the flexibility both add and subtract weights. If your operator is *idempotent* (that is, $a \oplus a = a$) then there is no harm in counting the same item multiple times.

Reporting: Return a list containing all the points of $P \cap Q$.

Top-k: Report the (up to) k points of $P \cap Q$ based on weight.

Complexity Bounds: Observe that all of the above queries can be answered in $O(n)$ time trivially, by just running through all the points and testing one-by-one whether they lie within Q. You want to think of n as being huge (e.g., millions or more), but you want a query time that is much smaller (say tens to hundreds). Data structures for range searching are usually analyzed in terms of two quantities, space and query time. Ideally, one would like to achieve $O(n)$ space and $O(\log n)$ query time, since this matches the best you can expect in 1-dimensional space. If you cannot achieve this "gold standard," you might hope to do well with one criteria or the other. For example, the query time might be $O(\log n)$, but the space is $O(n^2)$. Alternatively, the space might be $O(n)$, but the query time is $O(\sqrt{n})$. (Remember that from the perspective of asymptotics, logarithmic query times are always better than polynomial times, thus $\log^c n$ is better than n^b for positive numbers c and b, not matter how large c or how small b.)

Ideally, the query time does not depend on the number of points that lie within the query range. It doesn't matter whether your range contains no point or all the points, the (worst-case) running time depends only on n. The exception is reporting queries, since you need to take time to place the elements in the list. For example, the query time for range reporting might be expressed as $O(k + \log n)$, where $k = |P \cap Q|$ and $n = |P|$.

Another issue is the amount of time that it takes to construct the data structure. Ideally, the construction time should be about the same as the space, perhaps larger by a factor of $O(\log n)$.

Orthogonal Range Queries: In this lecture we will focus on perhaps the most common range search, called an *orthogonal range searching*. In this case a range is defined by axis-parallel rectangles in $\mathbb{R}^2$ and d-dimensional hyperrectangles in $\mathbb{R}^d$. An important property of orthogonal ranges is that they can be expressed as the product of 1-dimensional ranges. Let's assume that a point $p \in \mathbb{R}^d$ is expressed as its coordinate vector $(p_1, \ldots, p_d)$. Given two points $a, b \in \mathbb{R}^d$, where $a_i < b_i$ for $1 \leq i \leq d$, they define an axis-parallel rectangle (see Fig. 1)

$$R(a,b) \;=\; \{p \in \mathbb{R}^d : a_i \leq p_i \leq b_i\} \;=\; [a_1, b_1] \times \cdots \times [a_d, b_d]$$

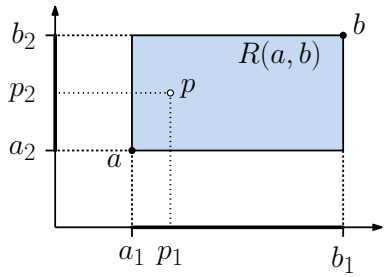

Fig. 1: Orthogonal range query.

Canonical Subsets: A common approach used in solving almost all range queries is to represent P as a collection of *canonical subsets* $\{P_1, \ldots, P_k\}$, each $P_i \subseteq P$ (where k is generally a function of n and the type of ranges), such that any set can be formed as the disjoint union of canonical subsets. Note that these subsets may generally overlap each other.

There are many ways to select canonical subsets, and the choice affects the space and time complexities. For example, the canonical subsets might be chosen to consist of n singleton sets, each of the form $\{p_i\}$. This would be very space efficient, since we need only $O(n)$ total space to store all the canonical subsets, but in order to answer a query involving k objects we would need k sets. (This might not be bad for reporting queries, but it would be too long for counting queries.) At the other extreme, we might let the canonical subsets be all the sets of the range space $\mathcal{R}$. Thus, any query could be answered with a single canonical subset (assuming we could determine which one), but we would have $|\mathcal{R}|$ different canonical subsets to store, which is typically a higher ordered polynomial in n, and may be too high to be of practical value. The goal of a good range data structure is to strike a balance between the total number of canonical subsets (space) and the number of canonical subsets needed to answer a query (time).

Perhaps the most common way in which to define canonical subsets is through the use of a *partition tree*. A partition tree is a rooted (typically binary) tree, whose leaves correspond to the points of P. Each node u of such a tree is naturally associated with a subset of P, namely, the points stored in the leaves of the subtree rooted at u. We will see an example of this when we discuss one-dimensional range queries.

One-dimensional range queries: Before we consider how to solve general range queries, let us consider how to answer 1-dimension range queries, or *interval queries*. Let us assume that we are given a set of points $P = \{p_1, p_2, \ldots, p_n\}$ on the line, which we will preprocess into a data

structure. Then, given an interval $[x_{lo}, x_{hi}]$, the goal is to count or report all the points lying within the interval. Ideally, we would like to answer counting queries in $O(\log n)$ time, and we would like to answer reporting queries in time $O((\log n) + k)$ time, where k is the number of points reported.

Clearly one way to do this is to simply sort the points, and apply binary search to find the first point of P that is greater than or equal to x_{lo}, and less than or equal to x_{hi}, and then enumerate (or count) all the points between. This works fine in dimension 1, but does not generalize readily to any higher dimensions. Also, it does not work when dealing with the weighted version, unless the weights are drawn from a group.

Let us consider a different approach, which will generalize to higher dimensions. Sort the points of P in increasing order and store them in the leaves of a balanced binary search tree. Each internal node of the tree is labeled with the largest key appearing in its left child. We can associate each node of this tree (implicitly or explicitly) with the subset of points stored in the leaves that are descendants of this node. This gives rise to the $O(n)$ *canonical subsets*. In order to answer reporting queries, the canonical subsets do *not* need to be stored explicitly with each node of the tree. The reason is that we can enumerate each canonical subset in time proportional to its size by simply traversing the subtree and reporting the points lying in its leaves. This is illustrated in Fig. 2. For range counting, we associate each node with the total weight of points in its subtree.

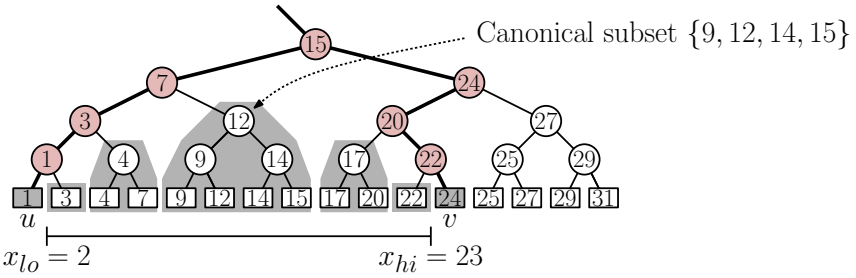

Fig. 2: Canonical sets for interval queries. For range reporting, canonical subsets are generated as needed by traversing the subtree.

We claim that the canonical subsets corresponding to any range can be identified in $O(\log n)$ time from this structure. Given any interval $[x_{lo}, x_{hi}]$, we search the tree to find the rightmost leaf u whose key is less than x_{lo} and the leftmost leaf v whose key is greater than x_{hi}. (To make this possible for all ranges, we could add two sentinel points with values of $-\infty$ and $+\infty$ to form the leftmost and rightmost leaves.) Clearly all the leaves between u and v constitute the points that lie within the range. To form these canonical subsets, we take the subsets of all the *maximal subtrees* lying between the paths from the root u and v.

Here is how to compute these subtrees. The search paths to u and v may generally share some common subpath, starting at the root of the tree. Once the paths diverge, as we follow the left path to u, whenever the path goes to the left child of some node, we add the canonical subset associated with its right child. Similarly, as we follow the right path to v, whenever the path goes to the right child, we add the canonical subset associated with its left child.

As mentioned earlier, to answer a range reporting query we simply traverse the canonical

subtrees, reporting the points of their leaves. To answer a range counting query we return the sum of weights associated with the nodes of the canonical subtrees.

Since the search paths to u and v are each of length $O(\log n)$, it follows that $O(\log n)$ canonical subsets suffice to represent the answer to any query. Thus range counting queries can be answered in $O(\log n)$ time. For reporting queries, since the leaves of each subtree can be listed in time that is proportional to the number of leaves in the tree (a basic fact about binary trees), it follows that the total time in the search is $O((\log n) + k)$, where k is the number of points reported.

In summary, 1-dimensional range queries can be answered in $O(\log n)$ (counting) or $((\log n) + k)$ (reporting) time, using $O(n)$ storage. This concept of finding maximal subtrees that are contained within the range is fundamental to all range search data structures. The only question is how to organize the tree and how to locate the desired sets. Let see next how can we extend this to higher dimensional range queries.

Kd-trees: The natural question is how to extend 1-dimensional range searching to higher dimensions. First we will consider kd-trees. This data structure is easy to implement and quite practical and useful for many different types of searching problems (nearest neighbor searching for example). However it is not the asymptotically most efficient solution for the orthogonal range searching, as we will see later.

Our terminology is a bit nonstandard. The data structure was designed by Jon Bentley. In his notation, these were called "k-d trees," short for "k-dimensional trees". The value k was the dimension, and thus there are 2-d trees, 3-d trees, and so on. However, over time, the specific value of k was lost. Our text uses the term "kd-tree" rather than "k-d tree." By the way, there are many variants of the kd-tree concept. We will describe the most commonly used one, which is quite similar to Bentley's original design. In our trees, points will be stored only at the leaves. There are variants in which points are stored at internal nodes as well.

A kd-tree is an example of a partition tree. For each node, we subdivide space either by splitting along the x-coordinates or along the y-coordinates of the points. Each internal node t of the kd-tree is associated with the following quantities:

$t.cut\text{-}dim$	the cutting dimension (e.g., $x = 0$ and $y = 1$)
$t.cut\text{-}val$	the cutting value (a real number)
$t.weight$	the number (or generally, total weight) of points in t's subtree

In dimension d, the cutting dimension may be represented as in integer ranging from 0 to $d - 1$. If the cutting dimension is i, then all points whose ith coordinate is less than or equal to $t.cut\text{-}val$ are stored in the left subtree and the remaining points are stored in the right subtree (see Fig. 3). If a point's coordinate is equal to the cutting value, then we may allow the point to be stored on either side. This is done to allow us to balance the number of points in the left and right subtrees if there are many equal coordinate values. When a single point remains (or more generally a small constant number of points), we store it in a leaf node, whose only field $t.point$ is this point.

The cutting process has a geometric interpretation. Each node of the tree is associated implicitly with a rectangular region of space, called a *cell*. (In general these rectangles may

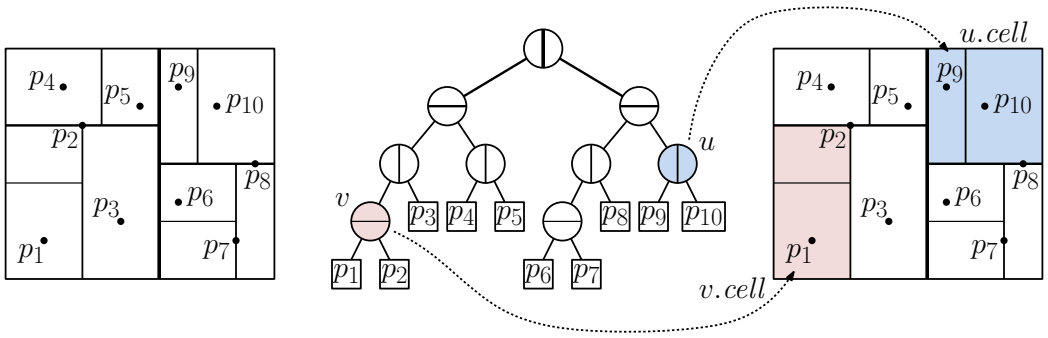

Fig. 3: (a) Point set and kd-subdivision, (b) kd-tree structure, and (c) example of node cells.

be unbounded, but in many applications it is common to restrict ourselves to some bounded rectangular region of space before splitting begins, and so all these rectangles are bounded.) The cells are nested in the sense that a child's cell is contained within its parent's cell. Hence, these cells define a *hierarchical decomposition* of space (see Fig. 3(a)).

There are two key decisions in the design of the tree.

How is the cutting dimension chosen? The simplest method is to cycle through the dimensions one by one. (This method is shown in Fig. 3.) Since the cutting dimension depends only on the level of a node in the tree, one advantage of this rule is that the cutting dimension need not be stored explicitly in each node, instead we keep track of it while traversing the tree.

One disadvantage of this splitting rule is that, depending on the data distribution, this simple cyclic rule may produce very skinny (elongated) cells, and such cells may adversely affect query times. Another method is to select the cutting dimension to be the one along which the points have the greatest *spread*, defined to be the difference between the largest and smallest coordinates. Bentley call the resulting tree an *optimized kd-tree*.

How is the cutting value chosen? To guarantee that the tree has height $O(\log n)$, the best method is to let the cutting value be the median coordinate along the cutting dimension. If there is an even number of points in the subtree, we may take either the upper or lower median, or we may simply take the midpoint between these two points. In our example, when there are an odd number of points, the median is associated with the left (or lower) subtree.

A kd-tree is a special case of a more general class of hierarchical spatial subdivisions, called *binary space partition trees* (or *BSP trees*) in which the splitting lines (or hyperplanes in general) may be oriented in any direction.

Constructing the kd-tree: It is possible to build a kd-tree in $O(n \log n)$ time by a simple top-down recursive procedure. The most costly step of the process is determining the median coordinate for splitting purposes. One way to do this is to maintain two lists of pointers to the points, one sorted by x-coordinate and the other containing pointers to the points sorted according to their y-coordinates. (In dimension d, d such arrays would be maintained.) Using

these two lists, it is an easy matter to find the median at each step in constant time. In linear time it is possible to split each list about this median element.

For example, if $x = s$ is the cutting value, then all points with $p_x \leq s$ go into one list and those with $p_x > s$ go into the other. (In dimension d this generally takes $O(d)$ time per point.) This leads to a recurrence of the form $T(n) = 2T(n/2) + n$, which solves to $O(n \log n)$. Since there are n leaves and each internal node has two children, it follows that the number of internal nodes is $n - 1$. Hence the total space requirements are $O(n)$.

Theorem: Given n points, it is possible to build a kd-tree of height $O(\log n)$ and space $O(n)$ in time $O(n \log n)$ time.

Range Searching in kd-trees: Let us consider how to answer orthogonal range counting queries. Range reporting queries are an easy extension. Let Q denote the desired range, and u denote the current node in the kd-tree. We assume that each node u is associated with its *cell*, denoted $u.cell$, which is the associated rectangular region in the hierarchical subdivision. (Cells can either be computed as part of preprocessing or computed on the fly as the algorithm is running.) The search algorithm is presented in the code block below.

kd-tree Range Counting Query

```
int range-count(Range Q, KDNode u)
(1) if (u is a leaf)
    (a) if (u.point ∈ Q) return u.weight,
    (b) else return 0 /* or generally, the semigroup identity element */
(2) else /* u is internal */
    (a) if (u.cell ∩ Q = ∅) return 0 /* the query does not overlap u's cell */
    (b) else if (u.cell ⊆ Q) return u.weight /* u's cell is contained within query range */
    (c) else, return range-count(Q, u.left) + range-count(Q, u.right).
```

The search algorithm traverses the tree recursively. If it arrives at a leaf cell, we check to see whether the associated point, $u.point$, lies within Q in $O(1)$ time, and if so we count it. Otherwise, u is an internal node. If $u.cell$ is disjoint from Q (which can be tested in $O(1)$ time since both are rectangles), then we know that no point in the subtree rooted at u is in the query range, and so there is nothing to count. If $u.cell$ is entirely contained within Q (again testable in $O(1)$ time), then every point in the subtree rooted at u can be counted. (These points constitute a canonical subset.) Otherwise, u's cell partially overlaps Q. In this case we recurse on u's two children and update the count accordingly.

Fig. 4 shows an example of a range search. Blue shaded nodes contribute to the search result and red shaded nodes do not. The red shaded subtrees are not visited. The blue-shaded subtrees are not visited for the sake of counting queries. Instead, we just access their total weight.

Analysis of query time: How many nodes does this method visit altogether? We claim that the total number of nodes is $O(\sqrt{n})$ assuming a balanced kd-tree. Rather than counting visited nodes, we will count nodes that are *expanded*. We say that a node is expanded if it is visited and both its children are visited by the recursive range count algorithm.

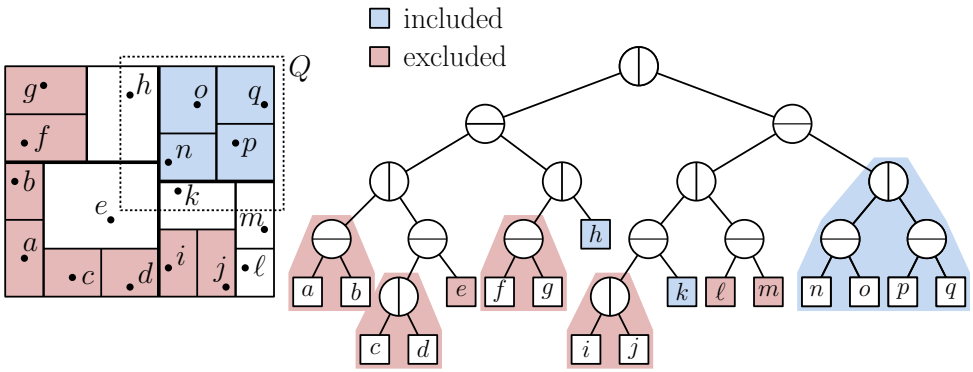

Fig. 4: Range search in a kd-tree. (Note: This particular tree was not generated by the algorithm described above.)

A node is expanded if and only if the cell overlaps the range without being contained within the range. We say that such a cell is *stabbed* by the query. To bound the total number of nodes that are expanded in the search, it suffices to bound the number of nodes whose cells are stabbed.

Lemma: Given a balanced kd-tree with n points using the alternating splitting rule, any vertical or horizontal line stabs $O(\sqrt{n})$ cells of the tree.

Proof: Let us consider the case of a vertical line $x = x_0$. The horizontal case is symmetrical.

Consider an expanded node which has a cutting dimension along x. The vertical line $x = x_0$ either stabs the left child or the right child but not both. If it fails to stab one of the children, then it cannot stab any of the cells belonging to the descendents of this child either. If the cutting dimension is along the y-axis (or generally any other axis in higher dimensions), then the line $x = x_0$ stabs both children's cells.

Since we alternate splitting on left and right, this means that after descending two levels in the tree, we may stab at most two of the possible four grandchildren of each node. In general each time we descend two more levels we double the number of nodes being stabbed. Thus, we stab the root node, at most 2 nodes at level 2 of the tree, at most 4 nodes at level 4, 8 nodes at level 6, and generally at most 2^i nodes at level $2i$. Each time we descend a level of the tree, the number of points falls by half. Thus, each time we descend two levels of the tree, the number of points falls by one fourth.

This can be expressed more formally as the following recurrence. Let $T(n)$ denote the number of nodes stabbed for a subtree containing n points. We have

$$T(n) \leq \begin{cases} 2 & \text{if } n \leq 4, \\ 1 + 2T\left(\frac{n}{4}\right) & \text{otherwise.} \end{cases}$$

We can solve this recurrence by appealing to the Master theorem for solving recurrences, as presented in the book by Cormen, Leiserson, Rivest and Stein. To keep the lecture self-contained, let's solve it by repeated expansion.

$$\begin{aligned} T(n) &\leq 1 + 2T\left(\frac{n}{4}\right) \\ &\leq 1 + 2\left(1 + 2T\left(\frac{n/4}{4}\right)\right) \;=\; (1+2) + 4T\left(\frac{n}{16}\right) \\ &\leq (1+2) + 4\left(1 + 2T\left(\frac{n/16}{4}\right)\right) \;=\; (1+2+4) + 8T\left(\frac{n}{64}\right) \\ &\leq \ldots \\ &\leq \sum_{i=0}^{k-1} 2^i + 2^k T\left(\frac{n}{4^k}\right). \end{aligned}$$

To get to the basis case ($T(1)$) let's set $k = \log_4 n$, which means that $4^k = n$. Observe that $2^{\log_4 n} = 2^{(\log_2 n)/2} = n^{1/2} = \sqrt{n}$. Since $T(1) \leq 2$, we have

$$T(n) \;\leq\; (2^{\log_4 n} - 1) + 2^{\log_4 n} T(1) \;\leq\; 3\sqrt{n} \;=\; O(\sqrt{n}).$$

This completes the proof.

We have shown that any vertical or horizontal line can stab only $O(\sqrt{n})$ cells of the tree. Thus, if we were to extend the four sides of Q into lines, the total number of cells stabbed by all these lines is at most $O(4\sqrt{n}) = O(\sqrt{n})$. Thus the total number of cells stabbed by the query range is $O(\sqrt{n})$. Since we only make recursive calls when a cell is stabbed, it follows that the total number of expanded nodes by the search is $O(\sqrt{n})$, and hence the total number of visited nodes is larger by just a constant factor.

Theorem: Given a balanced kd-tree with n points, orthogonal range counting queries can be answered in $O(\sqrt{n})$ time and reporting queries can be answered in $O(\sqrt{n} + k)$ time. The data structure uses space $O(n)$.