15. Range Trees

Lecture on Monday 24th November, 2008 by Bernd Gärtner <gaertner@inf.ethz.ch>

15.1 Window Queries

The following range searching problem is prototypical for applications in geographical information systems (GIS).

Problem 8 *Given a set* P *of* n *points in* $\mathbb{R}^2$*, we want to preprocess* P *into a data structure such that for any given* query window *(axis-parallel rectangle), the (number of) points in the window can quickly be reported, see Figure 15.1.*

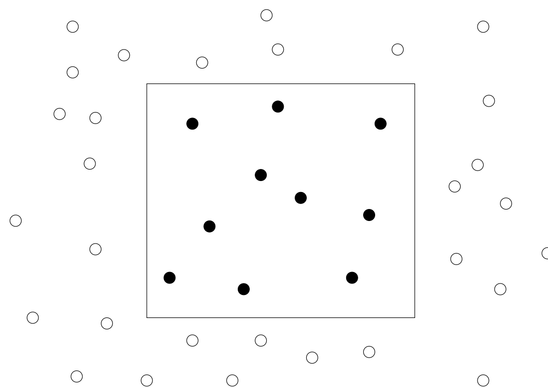

Figure 15.1: *Rectangular window query*

If we want just the number of points, we call this a *counting query*, and if we really want the list of points, this is a *reporting query*.

Think of a car navigator: the query window is a rectangle centered at the current position of the car, and we only want to "see" the small part of the world that is contained in the window.

If the point set P is very large (and this will be the case in GIS applications), it is not an option to go through the whole set P in each query in order to find out the points that are inside the query window. Instead, we are looking for an approach with logarithmic query time (plus the time needed to output the list of points, in case of a reporting query).

The 1-dimensional analog of this problem is the following: given a set S of n real numbers and an interval $[a, b]$, find all numbers of S in the interval. We are sure that you know how to solve this efficiently: the preprocessing step simply sorts S into ascending order and stores the sorted sequence in an array (the time required for this is $O(n \log n)$). Given an interval $[a, b]$, we then apply binary search to find the first element $\underline{s}$ in the sorted sequence that is greater or equal to a, and a second binary search to find the

last element $\overline{s}$ smaller or equal to b. Then the subsequence delimited by $\underline{s}$ and $\overline{s}$ is the desired output in a reporting query. In a counting query, we simply output one plus the difference of the array indices of $\overline{s}$ and $\underline{s}$.

The time to process a reporting query $[a, b]$ is $O(\log n + k)$, where k is the number of points that are being output. The $O(\log n)$ term comes from the two binary searches. For a counting query, the two binary searches and thus $O(\log n)$ query time suffices.

Our goal in the following is to generalize the binary search approach to the 2-dimensional setting.

15.2 The Range Tree

Consider $P \subseteq \mathbb{R}^2$, $|P| = n$. In a first step, we sort P into ascending lexicographic order and store the sorted sequence in an array. Over this array, we then erect a balanced binary tree with the n array elements as its leaves, see Figure15.2. (We don't have to build this tree explicitly, but it eases the exposition.)

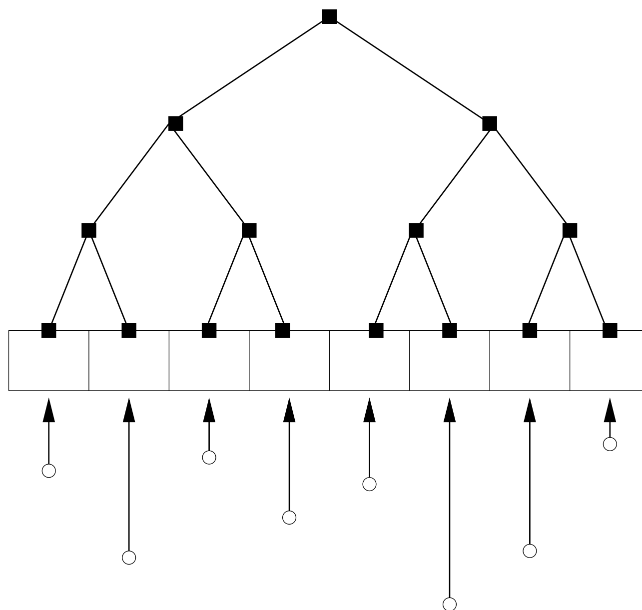

Figure 15.2: *First level of the range tree: A balanced binary tree represents the lexicographic order of points.*

Each node v of the tree naturally corresponds to a subset $P(v)$ of points, namely the points stored in the leaves of the subtree hanging off v. For example, if v is the root of the tree, then $P(v) = P$, and if v is a leaf, then $P(v)$ is a singleton.

In a second step, we store with each node v its set $P(v)$, where we represent $P(v)$ as an array, now sorted by y-coordinate (see Figure 15.3).

How large is this data structure? The first level surely requires $O(n)$ space, but the second level needs more: every point p appears in the sets $P(v)$ of all nodes along the

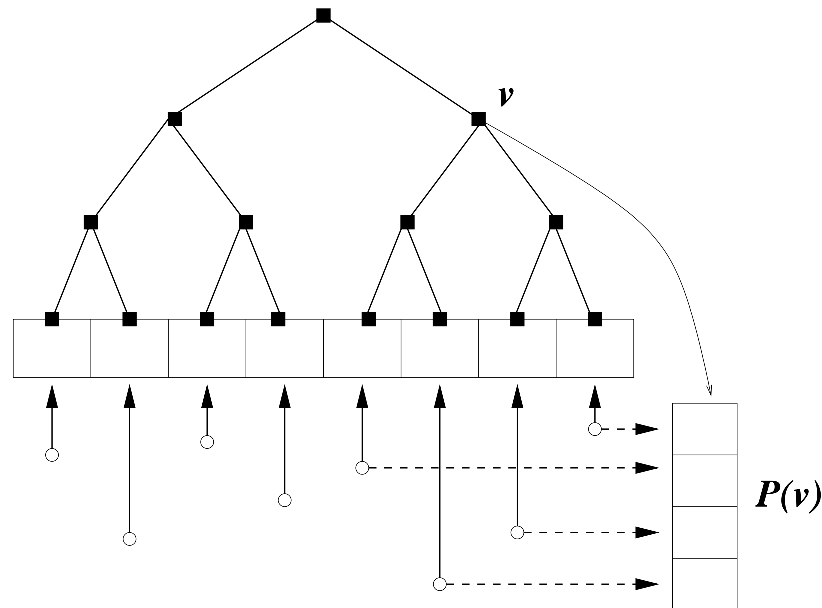

Figure 15.3: *Second level of the range tree: Each internal node v stores the points in its subtree sorted by y-coordinate (ties broken arbitrarily).*

path from p to the root. Since we have a balanced binary tree, each such path has length $O(\log n)$, meaning that the second level arrays require $O(n \log n)$ space in total. This also bounds the space requirements of the whole range tree.

As an exercise we ask you to prove that the range tree can also be built in $O(n \log n)$ time so that we obtain the following

Theorem 15.1 *Given n points in $\mathbb{R}^2$, we can preprocess them into a range tree in time $O(n \log n)$.*

15.3 The Query

Given a query rectangle $[x_{\min}, x_{max}] \times [y_{\min}, y_{\max}]$, we want to find all points $p \in P$ that are contained in that rectangle.

15.3.1 Finding the points with the right x-coordinates

A necessary condition for containment is that p is lexicographically *at least* as large as $(x_{\min}, y_{\min})$ and *at most* as large as $(x_{\max}, y_{\max})$. We can find the points with this property as in the 1-dimensional case: use two binary searches in the lexicographically sorted sequence to find the first point $\underline{p}$ lexicographically greater or equal to $(x_{\min}, y_{\min})$ and the last point $\overline{p}$ lexicographically smaller or equal to $(x_{\max}, y_{\max})$. Let $[\underline{p}, \overline{p}]$ denote the subsequence of points delimited by $\underline{p}$ and $\overline{p}$ in the first-level array (see Figure 15.4).

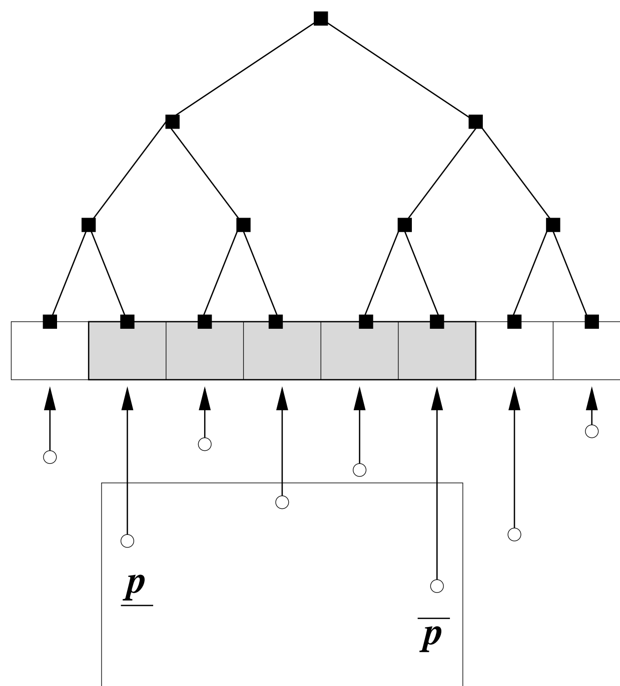

Figure 15.4: *Finding the points lexicographically between* (x_{min}, y_{min}) *(lower left corner of query rectangle) and* (x_{max}, y_{max}) *(upper right corner);* $\underline{p}$ *is the first such point,* $\overline{p}$ *the last, and* $[\underline{p}, \overline{p}]$ *(shaded area) is the subsequence of all such points.*

15.3.2 Finding the subset of points with the right y-coordinate

The subsequence $[\underline{p}, \overline{p}]$ definitely contains all points that are inside the query rectangle, but to correctly answer the query, we still need to report (or count) the ones among them with y-coordinate in $[y_{min}, y_{max}]$. For this, we use the secondary arrays, based on the following

Lemma 15.2 *The set* $[\underline{p}, \overline{p}]$ *is a disjoint union of* $O(\log n)$ *sets of the form* $P(v)$*, and these sets can be identified in time* $O(\log n)$*.*

Proof. There is a unique path in the tree that connects the leaf $\underline{p}$ to the leaf $\overline{p}$. To find this path, follow the two unique paths from $\underline{p}$ and $\overline{p}$ up to the root until they meet in a node u. Then, the path has the form

$$\underline{p} = \underline{u}_0, \ldots, \underline{u}_k, u, \overline{u}_k, \ldots, \overline{u}_0 = \overline{p},$$

see Figure 15.5. Let Π be the set of path vertices.

Now let W be the set of nodes

$$W = \{v \mid v \text{ is right child of some } \underline{u}_i, v \notin \Pi\} \cup \{v \mid v \text{ is left child of some } \overline{u}_i, v \notin \Pi\}.$$

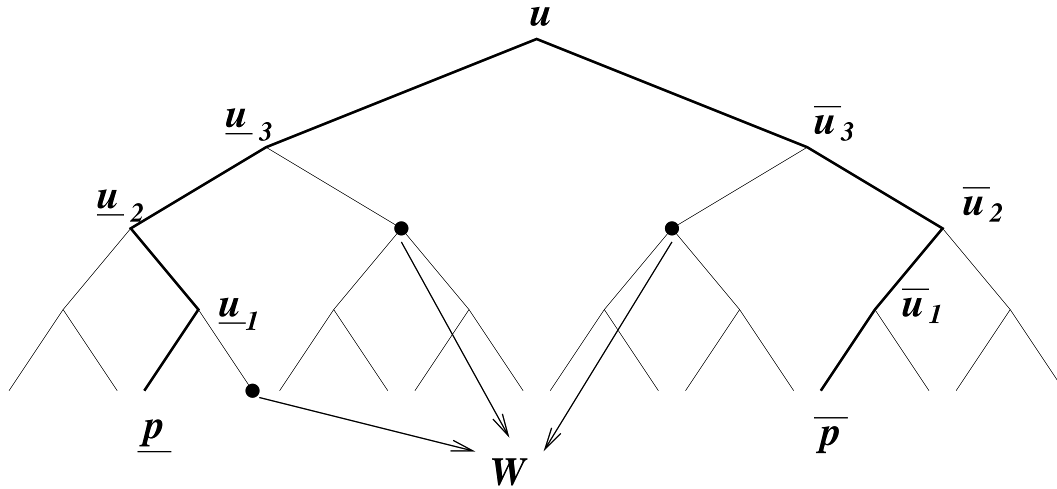

Figure 15.5: *The path in the tree connecting the leaves $\underline{p}$ and $\overline{p}$, and the set W of vertices "hanging off" the path from below*

These are the nodes "hanging off" the path Π from below, see Figure 15.5.

We now claim that

$$[\underline{p}, \overline{p}] = \{\underline{p}, \overline{p}\} \cup \bigcup_{v \in W} P(v) = P(\underline{p}) \cup P(\overline{p}) \cup \bigcup_{v \in W} P(v). \tag{15.3}$$

If we can prove this, we are done, since $|W| = O(\log n)$ (clearly, W can also be computed in this time, since we can find Π in time $O(\log n)$).

Let us first prove that each $p \in [\underline{p}, \overline{p}]$ is in the right-hand side union of (15.3). This is clear for $p \in \{\underline{p}, \overline{p}\}$. Otherwise, start at p and walk up to the root. Let v be the vertex just before hitting Π (we must eventually hit Π since p is between $\underline{p}$ and $\overline{p}$).

We clearly have $p \in P(v)$ and $v \notin \Pi$. If the parent of v is of the form $\underline{u}_i$, then v must be its right child, since otherwise, all vertices in $P(v)$ and therefore also p would be to the left of $\underline{p}$. Similarly, if the parent of v is of the form $\overline{u}_i$, then v must be its left child. It follows that $v \in W$.

In the other direction, let p be any leaf in the right-hand side union of (15.3). If $p \in \{\underline{p}, \overline{p}\}$, there is nothing to show; otherwise let $v \in W$ be the vertex such that $p \in P(v)$. If the parent of v is of the form $\underline{u}_i$, then p is to the left of $\overline{p}$ since we turned left at u in going from the root to p. On the other hand, p is also to the right of $\underline{p}$, since we continue to the right at $\underline{u}_i$ in going to p. Together this means that $p \in [\underline{p}, \overline{p}]$. The argument for v being a child of some $\overline{u}_i$ is symmetric. □

Using Lemma 15.2, the complete query proceeds in the following two steps.

1. Determine the interval $[\underline{p}, \overline{p}]$ of points that are lexicographically between (x_{min}, y_{min}) and (x_{max}, y_{max}).

2. For each $v \in W \cup \{\underline{p}, \overline{p}\}$, output (or just count) the points in $P(v)$ with y-coordinate in $[y_{\min}, y_{\max}]$.

For each node v considered in step 2, the selection from $P(v)$ can be made in time $O(\log n + k)$, where k is the number of points being output. This is again a 1-dimensional instance of the problem. Summing this up over the $O(\log n)$ nodes v being considered, we obtain the following

Theorem 15.4 *Given the range tree for a set* P *of* n *points in* $\mathbb{R}^2$*, a window reporting query can be answered in time* $O(\log^2 n + k)$*, where* k *is the number of points in the window. For a window counting query, the time is* $O(\log^2 n)$*.*

15.4 Query time $O(\log n)$ through Fractional Cascading

Compared to the 1-dimensional version of the problem, we have an additional log-factor in the query time. Is this unavoidable, or can we get the query time down to $O(\log n)$? Yes, we can!

The idea is to install some additional pointers that interconnect the $P(v)$'s. The property that we want to achieve is the following.

Property 15.5 *Let* v *be any tree node, and* v' *a child of it. If we know the subsequence of points in* $P(v)$ *with* y*-coordinate in* $[y_{\min}, y_{\max}]$*, it takes time* $O(1)$ *to identify the subsequence of points in* $P(v')$ *with* y*-coordinate in* $[y_{\min}, y_{\max}]$*.*

Before we show how this property can be established, let us draw the conclusion.

Theorem 15.6 *Given the range tree for a set* P *of* n *points in* $\mathbb{R}^2$*, enhanced in such a way that it satisfies Property 15.5. Then a window reporting query can be answered in time* $O(\log n + k)$*, where* k *is the number of points in the window. For a window counting query, the time is* $O(\log n)$*.*

Proof. Instead of searching each $P(v), v \in W$, individually, as before, we simply traverse the vertices of the path Π, from u down to $\underline{p}$, and to $\overline{p}$. In time $O(\log n)$, we can identify the subsequence of points in $P(u)$ with y-coordinate in $[y_{\min}, y_{\max}]$. Given this, we can do the same for any other vertex on Π in time $O(1)$ by Property 15.5. It follows that the sets $P(v), v \in W$, can also be searched for the relevant subsequences in time $O(1)$ per vertex, since every vertex in v is a child of some vertex on Π.

The time to identify the relevant subsequences in all sets $P(v), v \in W$ is therefore $O(\log n)$ (for the search in $P(u)$), plus another $O(\log n)$ (for $O(\log n)$ additional searches, each being handled in time $O(1)$). The total search cost is therefore $O(\log n)$, and this is the time needed to answer a window counting query. For the reporting query, we simply have to add k for outputting all the subsequences that have been identified. □

Now let's turn to how we establish Property 15.5. Consider Figure 15.6 for an example. It shows a vertex v with its sorted sequence $P(v)$ (here we assume that $P(v)$

only stores y-coordinates, as this is the only important information), along with its two children v' and v'' and their sorted sequences $P(v')$ and $P(v'')$. The important feature is that $P(v'), P(v'') \subseteq P(v)$.

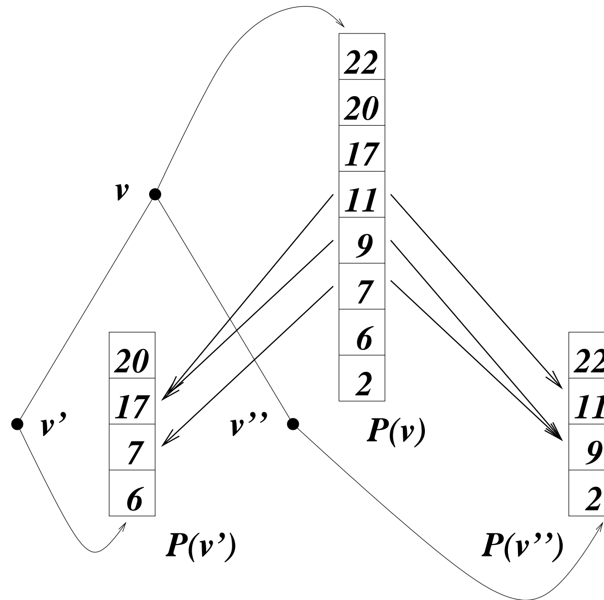

Figure 15.6: *Interconnecting the $P(v)$'s: every y in $P(v)$ points to the first elements in $P(v')$ and $P(v'')$ greater or equal to y.*

Every element y of $P(v)$ receives two additional pointers: one to the first element in $P(v')$ that is greater or equal to y, and another one to the first element in $P(v'')$ that is greater or equal to y (use a pointer past the end of the array if such an element does not exist, like for $y = 22$ and $P(v')$ in Figure 15.6).

Clearly, this increases the space requirements of the data structure by a constant factor only. At the same time, it achieves Property 15.5: assume that we know the subsequence $[\underline{y}, \overline{y}] \subseteq P(v)$ of y-coordinates in $[y_{\min}, y_{\max}]$. Here is how we find the first element of the relevant subsequence $[\underline{y}', \overline{y}']$ in $P(v')$, say.

Observation 15.7 *Let $\underline{y}$ be the first element in $P(v)$ greater or equal to $y_{\min}$, and let $\underline{y}' \in P(v')$ be the element pointed to by $\underline{y}$ (if any). Then $\underline{y}'$ is also the first element in $P(v')$ greater or equal to $y_{\min}$.*

Proof. There can't be any value z of $P(v')$ in $[y_{\min}, \underline{y}')$, since $z \in P(v') \subseteq P(v)$ would imply $\underline{y} \leq z < \underline{y}'$ which would in turn imply that $\underline{y}$ does not point to $\underline{y}'$ (but to z or some still smaller value). □

To find the last element of the relevant subsequence in $P(v')$, we use the following

Observation 15.8 *Let $\overline{y}$ be the last element in $P(v)$ smaller or equal to $y_{\max}$, and let $\overline{y}' \in P(v')$ be the element pointed to by $\overline{y}$ (if any). If $\overline{y}' \leq y_{\max}$, then $\overline{y}'$ is also the*

last element of $P(\nu')$ *smaller or equal to* $y_{\max}$. *Otherwise, the* predecessor *of* $\overline{y}'$ *in* $P(\nu')$[1] *is the last element of* $P(\nu')$ *smaller or equal to* $y_{\max}$.

Proof. If $\overline{y}' \leq y_{\max}$, there can't be any value z of $P(\nu')$ in $(\overline{y}', y_{\max}]$, since $z \in P(\nu') \subseteq P(\nu)$ would imply $\overline{y} \geq z > \overline{y}'$, a contradiction to $\overline{y}$ pointing to $\overline{y}'$.

Otherwise, let y' be the predecessor of $\overline{y}'$ in $P(\nu')$. By definition of $\overline{y}'$, y' is the last element of $P(\nu')$ that is smaller than $\overline{y}$. We are done if we can prove that there is also no value z of $P(\nu')$ in $[\overline{y}, y_{\max}]$. But this holds, since the smallest possible value greater or equal to $\overline{y}$ is $\overline{y}' > y_{\max}$. □

You might wonder why this technique of installing some additional pointers is called "fractional cascading", since there is nothing fractional and nothing cascading about it. Still, the term is appropriate since this is a special instance of a more general technique that can be applied to speed up searches in multiple lists that are *not* necessarily subsets of each other.

Assume that L and L' are unrelated sorted lists, and that we want to speed up the search for the interval $L' \cap [a, b]$, given that we already know the interval $L \cap [a, b]$. Then the idea of fractional cascading is to insert a suitable fraction of elements of L into L' and connect them with pointers from L in order to get efficient starting points for the search in L' . That's where the "fractional" comes from. And the "cascading" aspect comes in since some of the newly inserted elements of L' could be further propagated to still other lists. The general framework in which this works is a graph in which every vertex has its own private sorted list, and we want to search all the lists of some connected subgraph.

[1] defined as the last element of $P(\nu')$ in case $\overline{y}'$ does not exist