Quadtrees - Hierarchical Grids

Sariel Har-Peled

November 10, 2005

598 - Approximation Algorithms in Geometry

1 Quadtrees - a simple point-location data-structure

Let \( P \) be a planar map. To be more concrete, let \( P \) be a partition of the unit square into triangles (i.e., a mesh). Since any simple polygon can be triangulated, \( P \) can represent any planar map, and its partition of the unit square into different regions, with different properties. For the sake of simplicity, assume that every vertex in \( P \) shares at most, say, nine triangles.

Let us assume that we want to preprocess \( P \) for point-location queries. Of course, there are data-structures that can do it with \( O(n \log n) \) preprocessing time, linear space, and logarithmic query time. Instead, let us consider the following simple solution (which in the worst case, can be much worse).

Build a tree \( T \), where the root corresponds to the unit square. Every node \( v \in T \) corresponds to a cell \( \Box_v \) (i.e., a square), and it has four children. The four children correspond to the four squares formed by splitting \( \Box_v \) into four equal size squares, by horizontal and vertical cuts. The construction is recursive, and we start from \( v = \text{root}_T \). As long as the current node intersects more than, say, nine triangles, we create its children nodes, and we call recursively on each child, with the list of input triangles that intersect its square. We stop at a leaf, if its “conflict-list” (i.e., list of triangles it intersects) is of size at most nine. We store this conflict-list in the leaf.

Given a query point \( q \), in the unit square, we can compute the triangle of \( P \) containing \( q \), by traversing down \( T \) from the root, repeatedly going into the child of the current node, whose square contains \( q \). We stop at soon as we reach a leaf, and then we scan the leaf conflict-list, and check which of the triangles contains \( q \).

Of course, in the worst case, if the triangles are long and skinny, this quadtree might have unbounded complexity. However, for reasonable inputs (say, the triangles are fat), then the quadtree would have linear complexity in the input size (see Exercise 5.1). The big advantage of quadtrees of course, is their simplicity. In a lot of cases, quadtree would be a sufficient solution, and seeing how to solve a problem using a quadtree might be a first insight into a problem.

1.1 Fast point-location in a quadtree

One possible interpretation of quadtrees is that they are a multi-grid representation of a point-set. In particular, given a node \( v \), with a square \( S_v \), which is of depth \( i \) (the root has depth zero), then the side length of \( S_v \) is \( 2^{-i} \), and it is a square in the grid \( G_{2^{-i}} \). In fact, we will refer to \( \ell(v) = -i \) as the level of \( v \). However, a cell in a grid has a unique ID made out of two integer numbers. Thus, a node \( v \) of a quadtree is uniquely defined by the triple \( \text{id}(v) = (\ell(v), \lfloor x/r \rfloor, \lfloor y/r \rfloor) \), where \((x,y)\) is any point in \( \Box_v \), and \( r = 2^{\ell(v)} \).

Furthermore, given a query point \( q \), and a desired level \( \ell \), we can compute the the ID of the quadtree cell of this level that contains \( q \) in constant time. Thus, this suggests a very natural
FastPntLocInner\( (T, q, lo, hi) \).  
\[
\begin{aligned}
\text{mid} & \leftarrow \lfloor (lo + hi)/2 \rfloor \\
v & \leftarrow \text{getNode}(T, q, \text{mid}) \\
\text{if } v = \text{null} & \text{ then return FastPntLocInner}(T, q, \text{lo}, \text{mid} - 1). \\
w & \leftarrow \text{Child}(v, q) \\
\text{if } w = \text{null} & \text{ then return } v \\
\text{return FastPntLocInner}(T, q, \text{mid} + 1, \text{hi})
\end{aligned}
\]

Figure 1: One can perform point-location in a quadtree \( T \) by calling \( \text{FastPntLocInner}(T, q, 0, \text{height}(T)) \).

algorithm for doing a point-location in a quadtree: Store all the IDs of nodes in the quadtree in a hash-table, and also compute the maximal depth \( h \) of the quadtree. Given a query point \( q \), we now have access to any node along the point-location path of \( q \) in \( T \), in constant time. In particular, we want to find the point in \( T \) where the point-location path “falls off” the quadtree. This we can find by performing a binary search for the dropping off point. Let \( \text{getNode}(T, q, d) \) denote the procedure that, in constant time, returns the node \( v \) of depth \( d \) in the quadtree \( T \) such that \( \Box_v \) contains the point \( q \). Given a query point \( q \), we can perform point-location in \( T \) by calling \( \text{FastPntLocInner}(T, q, 0, \text{height}(T)) \). See Figure 1 for the pseudo-code for \( \text{FastPntLocInner} \).

Lemma 1.1: Given a quadtree \( T \) of size \( n \) and of height \( h \), one can preprocess it in linear time, such that one can perform a point-location query in \( T \) in \( O(\log h) \) time. In particular, if the quadtree has height \( O(\log n) \) (i.e., it is “balanced”), then one can perform a point-location query in \( T \) in \( O(\log \log n) \) time.

2 Compressed Quadtrees: Range Searching Made Easy

Let \( P \) be a set of \( n \) points in the plane of spread \( \Phi = \Phi(P) \).

Definition 2.1: For a set \( P \) of \( n \) points in any metric space, let \( \Phi(P) = \frac{\max_{p,q \in P} \|pq\|}{\min_{p,q \in P, p \neq q} \|pq\|} \) be the spread of \( P \). In words, the spread of \( P \) is the ratio between the diameter of \( P \) and the distance between the two closest points. Intuitively, the spread tells us the range of distances that \( P \) posses.

One can build a quadtree for \( P \), storing the points of \( P \) in the leaves of \( P \), where one keep splitting a node as long as it contains more than one point of \( P \). During this recursive construction, if a leaf contains no points of \( P \), we save space by not creating this leaf, and instead creating a null pointer in the parent node for this child.

Lemma 2.2: Let \( P \) be a set of \( n \) points in the unit square, such that \( \text{diam}(P) = \max_{p,q \in P} \|pq\| \geq 1/2 \). Let \( T \) be a quadtree of \( P \) constructed over the unit square. Then, the depth of \( T \) is bounded by \( O(\log \Phi(P)) \), it can be constructed in \( O(n \log \Phi(P)) \) time, and the total size of \( T \) is \( O(n \log \Phi(P)) \).

Proof: The construction is done by a straightforward recursive algorithm. Let us bound the depth of \( T \). Consider any two points \( p, q \in P \), and observe that a node \( v \) of \( T \) of level \( u = \lceil \log \|pq\| \rceil - 1 \).
containing \( p \) must not contain \( q \) (we remind the reader that \( \lg n = \log_2 n \)). Indeed, the diameter of \( \Box_v \) is smaller than \( \sqrt{2}^u < \sqrt{2}||pq||/2 < ||pq|| \). Thus, \( \Box_v \) cannot contain both \( p \) and \( q \). In particular, any node of \( T \) of level \( r = -\lfloor \lg \Phi \rfloor - 1 \) can contain at most one point of \( P \), where \( \Phi = \Phi(P) \). Thus, all the nodes of \( T \) are of depth \( O(\log \Phi) \).

Since the construction algorithm spends \( O(n) \) time at each level, it follows that the construction time is \( O(n \log \Phi) \), and this also bounds the size of the quadtree \( T \).

The bounds of Lemma 2.2 are tight, as one can easily verify, see Exercise 5.2. But in fact, if you inspect a quadtree generated by Lemma 2.2, you would realize that there are a lot of nodes of \( T \) which are of degree one. Indeed, a node \( v \) of \( T \) has degree larger than one, only if it has two children, and let \( P_v \) be the subset of points of \( P \) stored in the subtree of \( v \). Such a node \( v \) splits \( P_v \) into, at least, two subsets and globally there can be only \( n - 1 \) such splitting nodes.

Thus, a quadtree \( T \) contains a lot of “useless” nodes. We can replace such a sequence of edges by a single edge. To this end, we will store inside each quadtree node \( v \), its square \( \Box_v \), and its level \( \ell(v) \). Given a path of vertices in the quadtree that are of degree one, we will replace them with a single edge, from the first node in the path, till the last vertex in this path (this is the first node of degree larger than one). We call the resulting tree a compressed quadtree. Since all internal nodes in the new compressed quadtree have degree larger than one, it follows that it has linear size (however, it still can have linear depth).

As an application for such a compressed quadtree, consider the problem of counting how many points are inside a query rectangle \( r \). We can start from the root of the quadtree, and recursively traverse it, going down a node only if its region intersects the query rectangle. Clearly, we will report all the points contained inside \( r \). Of course, we have no guarantee about the query time performance of the query, but in practice, this might be fast enough.

### 2.1 Efficient construction of compressed quadtrees

Let \( P \) be a set of \( n \) points in the unit square, with unbounded spread. We are interested in computing the compressed quadtree of \( P \). The regular algorithm for computing a quadtree when applied to \( P \) might required unbounded time. Modifying it so it requires only quadratic time is an easy exercise.

Instead, compute in linear time a disk \( D \) of radius \( r \), which contains at least \( n/10 \) of the points of \( P \), such that \( r \leq 2r_{\text{opt}}(P,n/10) \), where \( r_{\text{opt}}(P,n/10) \) denotes the radius of the smallest disk containing \( n/10 \) points. Computing \( D \) can be done in linear time, by a rather simpler algorithm (Lemma 6.1).

Let \( l = 2^{\lfloor \lg r \rfloor} \). Consider the grid \( G_l \). It has a cell that contains \( (n/10)/25 \) points (since \( D \) is covered by \( 5 \times 5 = 25 \) grid cells of \( G_l \), since \( l \geq r/2 \)), and no grid cell contains more than \( 5(n/10) \) points, by Lemma 6.2 (iii). Thus, compute \( G_l(P) \), and find the cell \( c \) containing the largest number of points. Let \( P_{\text{in}} \) be the points inside this cell \( c \), and \( P_{\text{out}} \) the points outside this cell. We know that \( |P_{\text{in}}| \geq n/250 \), and \( |P_{\text{out}}| \geq n/2 \). Next, compute the compressed quadtrees for \( P_{\text{in}} \) and \( P_{\text{out}} \), respectively, and let \( T_{\text{in}} \) and \( T_{\text{out}} \) denote the respective quadtrees. Since the cell of the root of \( T_{\text{in}} \) has side length which is a power of two, and it belongs to the grid \( G_l \), it follows that \( c \) represents a valid region, which can be a node in \( T_{\text{out}} \) (note that if it is a node in \( T_{\text{out}} \), then it is empty). Thus, we can do a point-location query in \( T_{\text{out}} \), and hang the root of \( T_{\text{in}} \) in the appropriate node of \( T_{\text{out}} \). This takes linear time (ignoring the time to construct \( T_{\text{in}} \) and \( T_{\text{out}} \)). Thus, the overall construction time is \( O(n \log n) \).

**Theorem 2.3** Given a set \( P \) of \( n \) points in the plane, one can compute a compressed quadtree of \( P \) in \( O(n \log n) \) deterministic time.
A square is a canonical square, if it is contained inside the unit square, it is a cell in a grid $G_r$, and $r$ is a power of two (i.e., it might correspond to a node in a quadtree). For reasons that would become clear later, we want to construct the quadtree out of list of quadtree nodes that must appear in the quadtree. Namely, we get a list of canonical grid cells that must appear in the quadtree (i.e., the level of the node, together with its grid ID).

**Lemma 2.4** Given a list $C$ of $n$ canonical squares, all lying inside the unit square, one can construct a compressed quadtree $T$ such that for any square $c \in C$, there exists a node $v \in T$, such that $\Box_v = c$. The construction time is $O(n \log n)$.

**Proof:** The construction is similar to Theorem 2.3. Let $P$ be a set of $n$ points, where $p_c \in P$, if $c \in C$, and $p_c$ is the center of $c$. Next, find, in linear time, a canonical square $C$ that contains at least $n/250$ points of $P$, and at most $n/2$ points of $P$. Let $U$ be the list of all squares of $C$ that contain $c$, let $C_{in}$ be the list of squares contained inside $c$, and let $C_{out}$ be the list of squares of $C$ that do not intersect the interior of $c$. Recursively, build a compressed quadtree for $C_{in}$ and $C_{out}$, denoted by $T_{in}$ and $T_{out}$, respectively.

Next, sort the nodes of $U$ in decreasing order of their level. Also, let $\pi$ be the point-location path of $c$ in $T_{out}$. Clearly, adding all the nodes of $U$ to $T_{out}$ is no more than performing a merge of $\pi$ together with the sorted nodes of $U$. Whenever we encounter a square of $U$ that does not have a corresponding node at $\pi$, we create this node, and insert it into $\pi$. Let $T_{out}'$ denote the resulting tree. Next, we just hang $T_{in}$ in the right place in $T_{out}'$. Clearly, the resulting quadtree has all the squares of $C$ as nodes.

As for the running time, we have $T(C) = T(C_{in}) + T(C_{out}) + O(n) + O(|U| \log |U|) = O(n \log n)$, since $|C_{out}| + |C_{in}| + |U| = n$ and $|C_{in}|, |C_{out}| \leq (249/250)n$.

### 2.2 Fingering a Compressed Quadtree - Fast Point Location

Let $T$ be a compressed quadtree of size $n$. We would like to preprocess it so that given a query point, we can find the lowest node of $T$ whose cell contains a query point $q$. As before, we can perform this by traversing down the quadtree, but this might require $\Omega(n)$ time. Since the range of levels of the quadtree nodes is unbounded, we can no longer use binary search on the levels of $T$ to answer the query.

Instead, we are going to use a rebalancing technique on $T$. Namely, we are going to build a balanced tree $T'$, which would have cross pointers (i.e., fingers) into $T$. The search would be performed on $T'$ instead of on $T$. In the literature, the tree $T$ is known as a finger tree.

**Definition 2.5** Let $T$ be a tree with $n$ nodes. A separator in $T$ is a node $v$, such that if we remove $v$ from $T$, we remain with a forest, such that every tree in the forest has at most $\lceil n/2 \rceil$ vertices.

**Lemma 2.6** Every tree has a separator, and it can be computed in linear time.

**Proof:** Consider $T$ to be a rooted tree, and initialize $v$ to be the root of $T$. We perform a walk on $T$. If $v$ is not a separator, then one of the children of $v$ in $T$ must have a subtree of $T$ of size $\geq \lceil n/2 \rceil$ nodes. Set $v$ to be this node. Continue in this walk, till we get stuck. The claim is that $v$ is the required node. Indeed, since we always go down, and the size of the subtree shrinks, we must get stuck. Thus, consider $w$ as the node we get stuck at. Clearly, the subtree of $w$ contains at least $\lceil n/2 \rceil$ nodes (otherwise, we would not set $v = w$). Also, all the subtrees of $w$ have size $\leq \lceil n/2 \rceil$, and the connected component of $T \setminus \{w\}$ containing the root contains at most $n - \lceil n/2 \rceil \leq \lceil n/2 \rceil$ nodes. Thus, $w$ is the required separator.
This suggests a natural way for processing a compressed quadtree for point-location queries. Find a separator \( v \in T \), and create a root node \( f_v \) for \( T' \) which has a pointer to \( v \); now recursively build finger trees to each tree of \( T \setminus \{v\} \), and hang them on \( w \). Given a query point \( q \), we traverse \( T' \), where at node \( f_v \in T' \), we check whether the query point \( q \in □_v \), where \( v \) is the corresponding node of \( T \). If \( q \notin □_v \), we continue the search into the child of \( f_v \), which corresponds to the connected component outside \( □_v \) that was hung on \( f_v \). Otherwise, we continue into the child that contains \( q \). This takes constant time per node. As for the depth for the finger tree \( T' \), observe 
\[
D(n) \leq 1 + D(⌈n/2⌉) = O(\log n).
\]
Thus, a point-location query in \( T' \) takes logarithmic time.

**Theorem 2.7** Given a compressed quadtree \( T \) of size \( n \), one can preprocess it in \( O(n \log n) \) time, such that given a query point \( q \), one can return the lowest node in \( T \) whose region contains \( q \) in \( O(\log n) \) time.

### 3 Balanced quadtrees, and good triangulations

The *aspect ratio* of a convex body is the ratio between its longest dimension and its shortest dimension. For a triangle \( △ = abc \), the aspect ratio \( A_{\text{ratio}}(△) \) is the length of the longest side divided by the height of the triangle on the longest edge.

**Lemma 3.1** Let \( φ \) be the smallest angle for a triangle. We have that 
\[
1/\sin φ \leq A_{\text{ratio}}(△) \leq 2/\sin φ.
\]

*Proof:* Consider the triangle \( △ = △abc \).

We have \( A_{\text{ratio}}(△) = c/h \). However, \( h = b \sin φ \), and since \( a \) is the shortest edge in the triangle (since it is facing the smallest angle), it must be that \( b \) is the middle length edge. As such, \( 2b \geq a + b \geq c \). Thus, \( A_{\text{ratio}}(△) \geq b/h = b/(b \sin φ) = 1/\sin φ \). And similarly, \( A_{\text{ratio}}(△) \leq 2b/h = 2b/(b \sin φ) = 2/\sin φ \).

Another natural measure of sharpness is the *edge ratio* \( E_{\text{ratio}}(△) \), which is the ratio between a triangle’s longest and shortest edges. Clearly, \( A_{\text{ratio}}(△) > E_{\text{ratio}}(△) \), for any triangle \( △ \). For a triangulation \( T \), we denote by \( A_{\text{ratio}}(T) \) the maximum aspect ratio of a triangle in \( T \). Similarly, \( E_{\text{ratio}}(T) \) denotes the maximum edge ratio of a triangle in \( T \).

**Definition 3.2** A *corner* of a quadtree cell is one of the four vertices of its square. The *corners* of the quadtree are the points that are corners of its cells. We say that the side of a cell is *split* if either of the neighboring boxes sharing it is split. A quadtree is *balanced* if any side of an unsplit cell may contain only one quadtree corner in its interior. Namely, adjacent leaves are either of the same level, or of adjacent levels.

**Lemma 3.3** Let \( P \) be a set of points in the plane, such that \( \text{diam}(P) = Ω(1) \) and \( Φ = Φ(P) \). Then, one can compute a (minimal size) balanced quadtree \( T \) of \( P \), in time \( O(n \log n + m) \) time, where \( m \) is the size of the output quadtree.

*Proof:* Compute a compressed quadtree \( T \) of \( P \) in \( O(n \log n) \) time. Next, we traverse \( T \), and replace every compressed edge of \( T \) by the sequence of quadtree nodes that defines it. To guarantee
the balance condition, we create a queue of the nodes of $T$, and store the nodes of $T$ in a hash table, with their IDs.

We handle the nodes in the queue, one by one. For a node $v$, we check whether the current adjacent nodes to $\Box v$ are balanced. Specifically, let $c$ be one of $\Box v$’s neighboring cells in the grid of $\Box v$, and let $c_p$ be the square containing $c$ in a grid one level up. We compute $\text{id}(c), \text{id}(c_p)$, and check if there is a node in $T$ with those IDs. If not, we create a node $w$ with region $c_p$ and $\text{id}(c_p)$, and recursively retrieve its parent (i.e., if it exists we retrieve it, otherwise, we create it), and hang $w$ from the parent node. We credit the work involved in creating $w$ to the output size. We add all the new nodes to the queue. We repeat the process till the queue is empty.

Since the algorithm never creates nodes smaller than the smallest cell in the original compressed quadtree, it follows that this algorithm terminates. It is also easy to argue by induction that any balanced quadtree of $P$ must contain all the nodes we created. Overall, the running time of the algorithm is $O(n \log n + m)$, since the work associated with any newly created quadtree node is constant.

**Definition 3.4** The **extended cluster** of a cell $c$ in a quadtree $T$ is the set of $5 \times 5$ neighboring cells of $c$ in the grid containing $c$, which are all the cells in distance $< 2l$ from $c$, where $l$ is the sidelength of $c$.

A quadtree $T$ over a point set $P$ is well-balanced, if it is balanced, and for every leaf node $v$ that contains a (single) point of $P$, we have the property that all the nodes of the extended cluster of $v$ are leaves in $T$ (i.e., none of them is split and has children), and they do not contain any other point of $P$. In fact, we will also require that for every non-empty node $v$, all the nodes of the extended cluster of $v$ are nodes in the quadtree.

**Lemma 3.5** Given a point set $P$ of $n$ points in the plane, one can compute a well-balanced quadtree of $P$ in $O(n \log n + m)$ time, where $m$ is the size of the output quadtree.

**Proof:** We compute a balanced quadtree $T$ of $P$. Next, for every leaf node $v$ of $T$ which contains a point of $P$, we verify that all its extended cluster are leaves of $T$. If any other of the nodes of the extended cluster of $v$ contains a point of $P$, we split $v$. If any of the extended cluster nodes is missing as a leaf, we insert it into the quadtree (with its ancestors if necessary). We repeat this process till we stop. Of course, during this process, we keep the balanced property valid, by adding necessary nodes. Clearly, all this work can be charged to newly created nodes, and as such takes linear time in the output size once the compressed quadtree is computed.

A well-balanced quadtree $T$ of $P$ provides for every point, a region (i.e., extended cluster) where it is well protected from other points. It is now possible to turn the partition of the plane induced by the leaves of $T$ into a triangulation of $P$.

We “warp” the quadtree framework as follows. Let $y$ be the corner nearest $x$ of the leaf of $T$ containing $x$; we replace $y$ by $x$ as a corner of the quadtree. Finally, we triangulate the resulting planar subdivision. Unwarped boxes are triangulated with isosceles right triangles by adding a point in the center. Only boxes with unsplit sides have warped corners; for these we choose the diagonal that gives better aspect ratio. Figure 2 shows a triangulation resulting from a variant of this method.

**Lemma 3.6** The method above gives a triangulation $\mathcal{Q}(P)$ with $A_{\text{ratio}}(\mathcal{Q}(P)) \leq 4$.

**Proof:** The right triangles used to triangulate the unwarped cells have aspect ratio 2. If a cell with side length $l$ is warped, we have two cases.
In the first case, the input point of \( P \) is inside the square of the original cell. Then we assume that the diagonal touching the warped point is chosen; otherwise, the aspect ratio can only be better than what we prove. Consider one of the two triangles formed, with corners the input point and two other cell corners. The maximum length hypotenuse is formed when the warped point is on its original location, and has length \( h = \sqrt{2}l \). The minimum area is formed when the point is in the center of the square, and has area \( a = l^2/4 \). Thus, the minimum height of such a triangle \( \triangle \) is \( \geq 2a/h \), and \( A_{\text{ratio}}(\triangle) \leq h/(2a/h) = h^2/2a = 4 \).

In the second case, the input point is outside the original square. Since the quadtree is well balanced, the new point \( y \) is somewhere inside a square of sidelength \( l \) centered at \( x \) (since we always move the closest leaf corner to the new point). In this case, we assume that the diagonal not touching the warped point is chosen. This divides the cell into an isosceles right triangle and another triangle. If the chosen diagonal is the longest edge of the other triangle, then one can argue as before, and the aspect ratio is bounded by 4. Otherwise, the longest edge touches the input point. The altitude is minimized when the triangle is isosceles with as sharp an angle as possible; see Figure 3. Using the notation of Figure 3, we have \( y = (l/2, \sqrt{7}l/2) \). Thus,

\[
\mu = \text{area}(\triangle wyz) = \frac{1}{2} \begin{vmatrix} 1 & 0 & l \\ l & 0 & -l \\ l/2 & (\sqrt{7}/2)l & 0 \end{vmatrix} = \frac{1}{2} \begin{vmatrix} l & 0 & -l \\ l/2 & (\sqrt{7}/2 - 1)l & 0 \end{vmatrix} = \frac{\sqrt{7} - 1}{4} l^2.
\]
We have $h\sqrt{2l}/2 = \mu$, and thus $h' = \sqrt{2}\mu/l = \sqrt{\frac{7}{2}}l$. The longest distance $y$ can be from $w$ is $\alpha = \sqrt{(1/2)^2 + (3/2)^2}t = (\sqrt{10}/2)l$. Thus, the aspect ratio of the new triangle is bounded by $\alpha/h' = (\sqrt{10}/2) / \sqrt{\frac{7}{2}} \approx 2.717 \leq 4$.

For a triangulation $T$, let $|T|$ denote the number of triangles of $T$. The Delaunay triangulation of a point set is the triangulation formed by all triangles defined by the points such that their circumscribing triangles are empty (the fact that this collection of triangles forms a triangulation requires a proof). Delaunay triangulations are extremely useful, and have a lot of useful properties. We denote by $\mathcal{DT}(P)$ the Delaunay triangulation of $P$.

**Lemma 3.7** There is a constant $c'$, independent of $P$, such that $|\mathcal{QT}(P)| \leq c' \sum_{\Delta \in \mathcal{DT}(P)} \log E_{\text{ratio}}(\Delta)$.

**Proof:** For this lemma, we modify the description of our algorithm for computing $\mathcal{QT}(P)$. We compute the compressed quadtree $T''$ of $P$, and we uncompress the edges by inserting missing cells. Next, we split a leaf of $T''$ if it has side length $\kappa$, it is not empty (i.e., it contains a point of $P$), and there is another point of $P$ of distance $\leq 2\kappa$ from it. We refer to such a node as being crowded. We repeat this, till there are no crowded leaves. Let $T'$ denote the resulting quadtree. We now iterate over all the nodes $v$ of $T'$, and insert all the nodes of the extended cluster of $v$ into $T'$. Let $T$ denote the resulting quadtree. It is easy to verify that $T$ is well-balanced, and identical to the quadtree generated by the algorithm of Lemma 3.5 (although it is unclear how to implement the algorithm described here efficiently).

Now, all the nodes of $T$ that were created when adding the extended cluster nodes can be charged to nodes of $T'$. Therefore we need only count the total number of crowded cells in $T'$.

Linearly many crowded cells have more than one child with points in them. It can happen at most linearly many times that a non-empty cell $c$ has a point of $P$ outside it of distance $2\kappa$ from it, which in the next level is in a cell non-adjacent to the children of $c$, where $\kappa$ is the side length of the cell, as this point becomes further away due to the shrinking sizes of cells as they split.

If a cell $b$ containing a point is split because an extended neighbor was split, but no extended neighbor contains any point, then, when either $b$ or $b$’s parent was split, a nearby point became farther away than $2\kappa$. Again, this can only happen linearly many times.

Finally a cell may contain two points, or several extended neighbor cells may contain points, and this situation may persist when the cells split. If splitting the children of the cell or of its neighbors separates the points, we can charge linear total work. Otherwise, let $Y$ be a maximal set of points in the union of cell $b$ and its neighbors, such that splitting $b$, its neighbors, or the children of $b$ and its neighbors does not further divide $Y$. Then some triangle of $\mathcal{DT}(P)$ connects two points $y_1$ and $y_2$ in $Y$ with a point $z$ outside $Y$.

Each split not yet accounted for occurs between the step when $Y$ is separated from $z$, and the step when $y_1$ and $y_2$ become more than $2\kappa$ units apart. These steps are at most $O(\log E_{\text{ratio}}(\Delta y_1 y_2 z))$ quadtree levels apart, so we can charge all the crowded cells caused by $Y$ to $\Delta y_1 y_2 z$. This triangle will not be charged by any other cells, because once we perform the splits charged to it all three points become far away from each other in the quadtree.

Therefore the number of crowded cells can be counted as a linear term, plus terms of the form $O(\log E_{\text{ratio}}(\Delta abc))$ for some Delaunay triangles $\Delta abc$. 

---

1To see that, observe that there must be an edge connecting a point $y_1 \in Y$ with a point $z \in P \setminus Y$ (since the triangulation is connected). Next, by going around $y_1$ and the points it is connected to, it is easy to observe that since $Y$ diameter is (considerably) smaller than the distances between $y_1$ and $z$, there must be an edge between $y_1$ and another point $y_2$ of $Y$ (for example, take $y_2$ to be the closest point in $Y$ to $y_1$). This edge, together with the edge before it in the ordering around $y_1$, form the required triangle.
Theorem 3.8 Given any point set $P$, we can find a triangulation $QT(P)$ such that each point of $P$ is a vertex of $QT(P)$ and $A_{\text{ratio}}(QT(P)) \leq 4$. There is a constant $c''$, independent of $P$, such that if $T$ is any triangulation containing the points of $P$ as vertices, $|QT(P)| \leq c'' |T| \log A_{\text{ratio}}(T)$.

In particular, any triangulation with constant aspect ratio containing $P$ is of size $\Omega(QT(P))$. Thus, up to a constant, $QT(P)$ is an optimal triangulation.

Proof: Let $Y$ be the set of vertices of $T$. Lemma 3.7 states that there is a constant $c$ such that $|QT(Y)| \leq c \sum_{\triangle \in DT(Y)} \log E_{\text{ratio}}(\triangle)$. The Delaunay triangulation has the property that it maximizes the minimum angle of the triangulation, among all triangulations of the point set $P$.

If $Y = P$, then using this maxminangle property, we have $A_{\text{ratio}}(T) \geq \frac{1}{2} A_{\text{ratio}}(DT(P)) \geq \frac{1}{2} E_{\text{ratio}}(DT(P))$, by Lemma 3.1. Hence

$$|QT(P)| \leq c \sum_{\triangle \in DT(P)} \log E_{\text{ratio}}(DT(P)) = c |T| E_{\text{ratio}}(DT(P)) \leq 2c |T| A_{\text{ratio}}(T).$$

Otherwise, $P \subset Y$. Imagine running our algorithm on point set $Y$, and observe that $|QT(P)| \leq |QT(Y)|$. By the same argument as above, $|QT(Y)| \leq c |T| \log A_{\text{ratio}}(T)$.

Corollary 3.9 $|QT(P)| = O(n \log A_{\text{ratio}}(DT(P)))$.

Corollary 3.9 is tight, as can be easily verified.

4 Bibliographical notes

The authoritative text on quadtrees is the book by Samet [Sam89]. The idea of using hashing in quadtrees in a variant of an idea due to Van Emde Boas, and is also used in performing fast lookup in IP routing (using PATRICIA tries which are one dimensional quadtrees [WVT97]), among a lot of other applications.

The algorithm described for the efficient construction of compressed quadtrees, is as far as I know new. The classical algorithms for computing compressed quadtrees efficiently achieve the same running time, but require considerably more careful implementation, and paying careful attention to details [CK95, AMN+98]. The idea of fingering a quadtree is from [AMN+98] (although their presentation is different than ours).

Balanced quadtree and good triangulations are due to Bern et al. [BEG94], and our presentation closely follows theirs. The problem of generating good triangulations had received considerable attention recently, as it is central to the problem of generating good meshes, which in turn are important for efficient numerical simulations of physical processes. The main technique used in generating good triangulations is the method of Delaunay refinement. Here, one computes the Delaunay triangulation of the point set, and inserts circumscribed centers as new points, for “bad” triangles. Proving that this method converges and generates optimal triangulations is a non-trivial undertaking, and is due to Ruppert [Rup93]. Extending it to higher dimensions, and handling boundary conditions make it even more challenging. However, in practice, the Delaunay refinement method outperforms the (more elegant and simpler to analyze) method of Bern et al. [BEG94], which easily extends to higher dimensions. Namely, the Delaunay refinement method generates good meshes with fewer triangles.

Furthermore, Delaunay refinement methods are slower in theory. Getting an algorithm to perform Delaunay refinement in the same time as the algorithm of Bern et al. is still open, although Miller [Mil04] got an algorithm with only slightly slower running time.
Very recently, Alper Üngör came up with a “Delaunay-refinement type” algorithm, which outputs better meshes than the classical Delaunay refinement algorithm [Ung04]. Furthermore, by merging the quadtree approach with Üngör technique, one can get an optimal running time algorithm [?].

5 Exercises

Exercise 5.1 [5 Points] A triangle $\triangle$ is called $\alpha$-fat if each one of its angles is at least $\alpha$, where $\alpha > 0$ is a prespecified constant (for example, $\alpha$ is 5 degrees). Let $P$ be a triangular planar map of the unit square (i.e., each face is a triangle), where all the triangles are fat, and the total number of triangles is $n$. Prove that the complexity of the quadtree constructed for $P$ is $O(n)$.

Exercise 5.2 [5 Points] Prove that the bounds of Lemma 2.2 are tight. Namely, show that for any $r > 2$ and any positive integer $n > 2$, there exists a set of $n$ points with diameter $\Omega(1)$ and spread $\Phi(P) = \Theta(r)$, and such that its quadtree has size $\Omega(n \log \Phi(P))$.

Acknowledgments

The author wishes to thank John Fischer for his detailed comments on the manuscript.

6 From previous lectures

Lemma 6.1 Given a set $P$ of $n$ points in the plane, and parameter $k$, one can compute in $O(n(n/k)^2)$ deterministic time, a circle $D$ that contains $k$ points of $P$, and radius($D$) $\leq 2r_{opt}(P,k)$.

Lemma 6.2 For any point set $P$, and $r > 0$, we have: (i) For any real number $A > 0$, it holds $\text{depth}(P, Ar) \leq (A + 1)^2 \text{depth}(P, r)$, (ii) $\text{gd}_r(P) \leq \text{depth}(P, r) \leq 9\text{gd}_r(P)$, (iii) if $r_{opt}(P,k) \leq r \leq 2r_{opt}(P,k)$ then $\text{gd}_r(P) \leq 5k$, and (iv) Any circle of radius $r$ is covered by at least one grid cluster in $G_r$.

References


