Chapter 2

Quadtrees - Hierarchical Grids

By Sariel Har-Peled, August 24, 2008[1]

In this chapter, we discuss quadtrees which is arguably one of the simplest and most powerful geometric data-structure. We begin in Section 2.1 by giving a simple application of quadtrees and describe a clever way for performing point-location queries quickly in such a quadtree. In Section 2.2, we describe how such quadtrees can be compressed and how can they be quickly constructed and used for point-location queries. In Section 2.3 we describe a randomized extension of this data-structure, known as *skip-quadtree*, which enables us to maintain the compressed quadtree efficiently under insertions and deletions. In Section 2.5, we turn our attention to applications of compressed quadtrees, showing how quadtrees can be used to compute good triangulations of an input point set.

2.1 Quadtrees - a simple point-location data-structure

Let P_{map} be a planar map. To be more concrete, let P_{map} be a partition of the unit square into triangles (i.e., a mesh). The partition P_{map} can represent any planar map, where a region in the map might be composed of several triangles. For the sake of simplicity, assume that every vertex in P_{map} shares at most, say, nine triangles.

Let us assume that we want to preprocess P_{map} 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 $\mathcal{T}$, where the root corresponds to the unit square. Every node $v \in T$ corresponds to a cell $\square_v$ (i.e., a square), and it has four children. The four children correspond to the four squares formed by splitting $\square_v$ 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.

[1]This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit `http://creativecommons.org/licenses/by-nc/3.0/` or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Given a query point q, in the unit square, we can compute the triangle of P_{map} containing q, by traversing down $\mathcal{T}$ 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 2.7.1) .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.

2.1.1 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 $\mathsf{G}_{2^{-i}}$. 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 $\square_v$, and $r = 2^{\ell(v)}$.

Furthermore, given a query point q, and a desired level ℓ, we can compute the ID of the quadtree cell of this level that contains q in constant time. Thus, this suggests a very natural 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 $\mathcal{T}$, in constant time. In particular, we want to find the point in $\mathcal{T}$ where the point-location path "falls off" the quadtree. This we can find by performing a binary search for the dropping off point. Let **QTGetNode**(T, q, d) denote the procedure that, in constant time, returns the node v of depth d in the quadtree $\mathcal{T}$ such that $\square_v$ contains the point q. Given a query point q, we can perform point-location in $\mathcal{T}$ by calling **QTFastPntLocInner**$(T, q, 0, \text{height}(T))$. See Figure 2.1 for the pseudo-code for **QTFastPntLocInner**.

```
QTFastPntLocInner(T, q, lo, hi).
    mid ← ⌊(lo + hi)/2⌋
    v ← QTGetNode(T, q, mid)
    if v = null then
        return QTFastPntLocInner(T, q, lo, mid − 1).
    w ← Child(v, q)
        //w is child of v containing the point q.
    If w = null then
        return v
    return QTFastPntLocInner(T, q, mid + 1, hi)
```

Figure 2.1: One can perform point-location in a quadtree $\mathcal{T}$ by calling **QTFastPntLocInner** $(T, q, 0, \text{height}(T))$.

Lemma 2.1.1 *Given a quadtree $\mathcal{T}$ of size n and of height h, one can preprocess it in linear time, such that one can perform a point-location query in $\mathcal{T}$ 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 $\mathcal{T}$ in $O(\log \log n)$ time.*

2.2 Compressed Quadtrees: Range Searching Made Easy

Definition 2.2.1 (Spread.) For a set P of n points in a metric space, let

$$\Phi(\mathsf{P}) = \frac{\max_{p,q\in\mathsf{P}} \|p-q\|}{\min_{p,q\in P, p\neq q} \|p-q\|}$$

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 $\mathcal{T}$ for P, storing the points of P in the leaves of $\mathcal{T}$, 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.

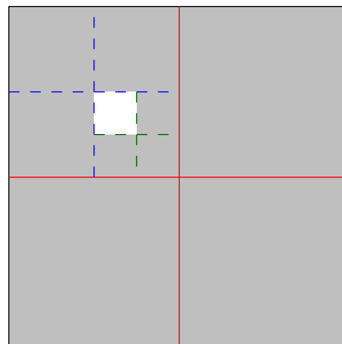
Lemma 2.2.2 *Let P be a set of n points in the unit square, such that* $\mathrm{diam}(\mathsf{P}) = \max_{p,q\in\mathsf{P}} \|p-q\| \geq 1/2$*. Let $\mathcal{T}$ be a quadtree of P constructed over the unit square. Then, the depth of $\mathcal{T}$ is bounded by $O(\log \Phi(\mathsf{P}))$, it can be constructed in $O(n \log \Phi(\mathsf{P}))$ time, and the total size of $\mathcal{T}$ is $O(n \log \Phi(\mathsf{P}))$.*

Proof: The construction is done by a straightforward recursive algorithm as described above.

Let us bound the depth of $\mathcal{T}$. Consider any two points $p, q \in P$, and observe that a node v of $\mathcal{T}$ of level $u = \lfloor \lg \|p-q\| \rfloor - 1$ containing p must not contain q (we remind the reader that $\lg n = \log_2 n$). Indeed, the diameter of $\square_v$ is smaller than $\sqrt{2}2^u < \sqrt{2}\|pq\|/2 < \|p-q\|$. Thus, $\square_v$ can not contain both p and q. In particular, any node of $\mathcal{T}$ 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 $\mathcal{T}$ are of depth $O(\log \Phi)$.

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

The bounds of Lemma 2.2.2 are tight, as one can easily verify, see Exercise 2.7.2. But in fact, if you inspect a quadtree generated by Lemma 2.2.2, you would realize that there are a lot of nodes of $\mathcal{T}$ which are of degree one (the degree of a node is the number of children it has). Indeed, a node v of $\mathcal{T}$ 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 $\mathcal{T}$ 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 $\square_v$, and its level $\ell(v)$. Given a path of vertices in the quadtree that are all of degree one, we will replace them with a single vertex that corresponds to the first vertex in this path, and its only child would be the last vertex in this path (this is the first node of degree larger than one). This ***compressed node*** has a single child, and the region rg_v that it is in "charge" of is an annulus, see the figure on the right. Otherwise, the ***region*** that a node is in charge of is a $\mathsf{rg}_v = \square_v$. The child corresponds to the inner square. We call the resulting tree a ***compressed quadtree***. Since any node that has only a single child is compressed, we can charge it to its parent, which has two children. Since there are at most $n-1$ internal nodes in the new compressed quadtree that have

degree larger than one, it follows that it has linear size (however, it still can have linear depth in the worst case).

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, but in practice, this might be fast enough.

2.2.1 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 $r \leq 2r_{\mathrm{opt}}(\mathsf{P}, n/10)$, where $r_{\mathrm{opt}}(\mathsf{P}, n/10)$ denotes the radius of the smallest disk containing $n/10$ points. Computing D can be done in linear time, by a rather simple algorithm (Lemma 2.8.1).

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 2.8.2 (iii). Thus, compute $\mathsf{G}_l(P)$, and find the cell c containing the largest number of points. Let P_{in} be the points inside this cell c, and P_{out} the points outside this cell. We know that $|P_{\mathrm{in}}| \geq n/250$, and $|P_{\mathrm{out}}| \geq n/2$. Next, compute the compressed quadtrees for P_{in} and P_{out}, respectively, and let $\mathcal{T}_{\mathrm{in}}$ and $\mathcal{T}_{\mathrm{out}}$ denote the respective quadtrees. Since the cell of the root of $\mathcal{T}_{\mathrm{in}}$ 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 $\mathcal{T}_{\mathrm{out}}$ (note that if it is a node in $\mathcal{T}_{\mathrm{out}}$, then it is empty). Thus, we can do a point-location query in $\mathcal{T}_{\mathrm{out}}$, and hang the root of $\mathcal{T}_{\mathrm{in}}$ in the appropriate node of $\mathcal{T}_{\mathrm{out}}$. This takes linear time (ignoring the time to construct $\mathcal{T}_{\mathrm{in}}$ and $\mathcal{T}_{\mathrm{out}}$). Thus, the overall construction time is $O(n \log n)$.

Theorem 2.2.3 *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.*

Definition 2.2.4 (Canonical square and grid.) 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). We will refer to such a grid G_r, as a ***canonical grid***.

For reasons that would become clear later, we want to construct the quadtree out of a 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 2.2.5 *Given a list C of n canonical squares, all lying inside the unit square, one can construct a compressed quadtree $\mathcal{T}$ such that for any square $c \in \mathsf{C}$, there exists a node $v \in T$, such that $\square_v = c$. The construction time is $O(n \log n)$.*

Proof: The construction is similar to Theorem 2.2.3. Let P be a set of n points, where $p_c \in \mathsf{P}$, if $c \in \mathsf{C}$, 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 $\mathcal{C}_{\text{in}}$ be the list of squares contained inside c, and let $\mathcal{C}_{\text{out}}$ be the list of squares of C that do not intersect the interior of c. Recursively, build a compressed quadtree for $\mathcal{C}_{\text{in}}$‘ and $\mathcal{C}_{\text{out}}$, denoted by $\mathcal{T}_{\text{in}}$ and $\mathcal{T}_{\text{out}}$, respectively.

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

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

2.2.2 Fingering a Compressed Quadtree - Fast Point Location

Let $\mathcal{T}$ 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 $\mathcal{T}$ 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 $\mathcal{T}$ to answer the query.

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

Definition 2.2.6 Let $\mathcal{T}$ be a tree with n nodes. A ***separator*** in $\mathcal{T}$ is a node v, such that if we remove v from $\mathcal{T}$, we remain with a forest, such that every tree in the forest has at most $\lceil n/2 \rceil$ vertices.

Lemma 2.2.7 *Every tree has a separator, and it can be computed in linear time.*

Proof: Consider $\mathcal{T}$ to be a rooted tree, and initialize v to be the root of $\mathcal{T}$. We perform a walk on $\mathcal{T}$. If v is not a separator, then one of the children of v in $\mathcal{T}$ must have a subtree of $\mathcal{T}$ of size $\ge \lceil n/2 \rceil$ 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 got 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 $\le \lceil n/2 \rceil$, and the connected component of $T \setminus \{w\}$ containing the root contains at most $n - \lceil n/2 \rceil \le \lfloor n/2 \rfloor$ 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 $\mathcal{T}'$ 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 $\mathcal{T}'$, where at node $f_v \in \mathcal{T}'$, we check whether the query point $q \in \square_v$, where v is the corresponding node of $\mathcal{T}$. If $q \notin \square_v$, we continue the search into the child of f_v, which corresponds to the connected component outside $\square_v$ 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 $\mathcal{T}'$, observe $D(n) \le 1 + D(\lceil n/2 \rceil) = O(\log n)$. Thus, a point-location query in $\mathcal{T}'$ takes logarithmic time.

Theorem 2.2.8 *Given a compressed quadtree $\mathcal{T}$ 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 $\mathcal{T}$ whose region contains q in $O(\log n)$ time.*

2.3 Dynamic Quadtrees

What if we want to maintain the compressed quadtree under insertions and deletions? There is an elegant way of doing this by using randomization. The resulting structure has similar behavior to skip-list where instead of linked list we use quadtrees.

We remind the reader the concept of gradation:

Definition 2.3.1 (*Gradation.*) Given a set P of n points, a *sampling sequence* $(S_m, \ldots, S_1)$ of P is a sequence of subsets of P, such that (i) $S_1 = \mathsf{P}$, (ii) S_i is formed by picking each point of S_{i-1} with probability $1/2$, and (iii) $|S_m| \leq 2k$, and $|S_{m-1}| > 2k$, where k is some prespecified constant. The sequence $(S_m, S_{m-1}, \ldots, S_1)$ is called a ***gradation*** of P.

Let $\mathcal{T}_1, \ldots, \mathcal{T}_m$ be the quadtrees of the sets $\mathsf{P} = S_1, \ldots, S_m$, respectively. Note, that the nodes of $\mathcal{T}_i$ are a subset of the nodes appear in $\mathcal{T}_{i-1}$. As such, every node in $\mathcal{T}_i$ would have pointers to its own copy in $\mathcal{T}_{i-1}$ and a pointer to its copy in $\mathcal{T}_{i+1}$ if it exists there. We will refer to this data-structure as ***skip-quadtree***.

Point-location queries. Given a query point q we want to find the leaf of $\mathcal{T}_1$ that contains it. The search algorithm is quite simple, starting at $\mathcal{T}_m$ you find the leaf in $\mathcal{T}_i$ that contains the query point, and then move to the corresponding node in $\mathcal{T}_{i-1}$, and continue the search from there.

2.3.1 Inserting a point into the skip-quadtree.

Let p be the point to be inserted into the skip-quadtree. We perform a point-location query and find the lowest node v in $\mathcal{T}_1$ that contains p. Next, we split v and establish a new node (hanging it from v) that contains p. Now, we flip an unbiased coin, if the coin comes up tail, we are done. Otherwise, we add p to $\mathcal{T}_2$. We continue in this fashion, adding p to the quadtrees in the relevant levels, till the coin comes up tail.

Note, that the amount of work and space needed at each level is a constant (ignoring the initial point-location query), and by implementing this operation carefully, the time to perform it would be proportional to the point-location query time.

Deleting a point from the skip-quadtree is done in a similar fashion to the insertion described above.

Given a point-set P, **constructing** the skip-quadtree can be done by inserting the points of P one by one into the skip-quadtree.

2.3.2 Running time analysis

We analyze the time needed to perform a point-location query. In particular, we claim that the expected query time $O(\log n)$. To see that we will use the standard backward analysis. Consider

the leaf v of $\mathcal{T}_i$ that contains q, and consider the path $v = v_1, v_2, \ldots, v_r$ from v to the root v_r of the compressed quadtree $\mathcal{T}_i$. Let π denote this path. Clearly, the amount of time spent in the search in the tree $\mathcal{T}_i$, is proportional to how far we have to go on this list, till we hit a node that appears in $\mathcal{T}_{i+1}$. (During the point-location we traverse this snippet of the path in the other direction - we are in a leaf of $\mathcal{T}_{i+1}$, we jump into the corresponding node of $\mathcal{T}_i$, and traverse down till we reach a leaf.) Note, that the node v_j stores (at least) j points of S_i, and if any pair of them appears in S_{i+1} then at least one of the nodes on π below the node v_j would appear in $\mathcal{T}_{i+1}$ (since the quadtree $\mathcal{T}_{i+1}$ needs to "separate" these two points and this is done by a node of the path). Let denote this set of points by U_j, and let X_j be an indicator variable which is one if v_j does not appear in $\mathcal{T}_{i+1}$. Clearly,

$$\mathbf{E}\Big[X_j\Big] = \mathbf{Pr}\Big[\text{no pair of points of } U_j \text{ is in } T_{i+1}\Big] \le \frac{j+1}{2^j},$$

since the points of S_i are randomly and independently chosen to be in S_{i+1} and the event happens only if zero or one points of U_j are in S_{i+1}.

Thus, the expected search time in T_i is $\mathbf{E}\Big[\sum_j X_j\Big] = \sum_j \mathbf{E}\Big[X_j\Big] = \sum_j (j+1)/2^j = O(1)$.

Thus, the overall expected search time in the skip-quadtree is proportional to the number of levels in the gradation.

Let $Z_i = |S_i|$ be the number elements stored in the ith level of the gradation. We know that $Z_1 = n$, and $\mathbf{E}[Z_i] =$. In particular, $\mathbf{E}[Z_i] = \mathbf{E}[\mathbf{E}[Z_i]] = \mathbf{E}[Z_{i-1}/2] = \cdots = n/2^{i-1}$. Thus, $\mathbf{E}[Z_\alpha] \le 1/n^{10}$, where $\alpha = \lceil 11 \lg n \rceil$. Thus, by Markov's inequality, we have that

$$\mathbf{Pr}[m > \alpha] = \mathbf{Pr}[Z_\alpha \ge 1] \le \frac{\mathbf{E}[Z_\alpha]}{1} = \frac{1}{n^{10}}.$$

We summarize:

Lemma 2.3.2 *A gradation defined over n elements has $O(\log n)$ levels both in expectation and with high probability.*

This implies that a point-location query in the skip quadtree takes, in expectation, $O(\log n)$ time.

Since, with high probability, there are only $O(\log n)$ levels in the gradation, it follows that the expected search time is $O(\log n)$.

High Probability. In fact, one can show that this point-location query time bound holds with high probability, and we sketch (informally) the argument why this is true. Consider the variable Y_i that is the number of nodes of T_i being visited during the point-location query. We have that $\mathbf{Pr}[Y_i \ge k] \le \sum_{j=k} (k+1)/2^k = O(k/2^k) = O(1/c^k)$, for some constant $c > 1$. Thus, we have a sum of logarithmic number of independent random variables, each one of them behaves like a variable with geometric distribution. As such, we can apply a Chernoff-type inequality (see Exercise 2.8.3) to get an upper bound on the probably that the sum of these variables exceeds $O(\log n)$. This probability is bounded by $1/n^{O(1)}$.

Note, the longest query time is realized by one of the points stored in the quadtree. Since there are n points stored in the quadtree, this implies that with high probability *all* point-location queries takes $O(\log n)$ time. Also, observe that the structure of the skip-quadtree is uniquely determined by the gradation. Since the gradation is oblivious to the history of the data-structure (i.e., what points

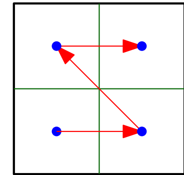
where inserted and deleted). As such, these bounds on the performance hold at any point in time during the usage of the skip-quadtree. We summarize:

Theorem 2.3.3 *Let $\mathcal{T}$ be an empty skip-quadtree used for a sequence of n operations (i.e., insertions, deletions and point-location queries). Then, with high probability (and thus also in expectation), the time to perform each such operation takes $O(\log n)$ time.*

2.4 Even More on Dynamic Quadtrees

The previous section reveals that quadtrees can be maintained dynamically using ideas similar to skip-lists. Here we will take this idea even further, showing how to facilitate quadtrees using any data-structure for ordered sets.

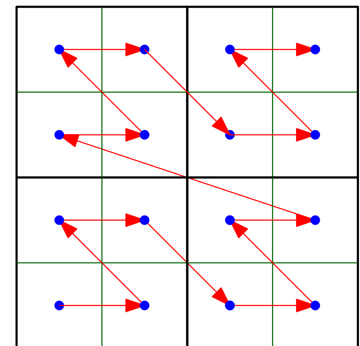
2.4.1 Ordering of nodes and points



So consider a quadtree $\mathcal{T}$, and a **DFS** traversal of $\mathcal{T}$, where the **DFS** always traverse the children of a node in the same relative order (i.e., say, first the bottom-left child, then the bottom-right child, top-left child, and top-right child).

Consider any two canonical squares $\square$ and $\widehat{\square}$, and imagine a quadtree $\mathcal{T}$ that contains both squares (i.e., there are nodes in $\mathcal{T}$ with these squares as their cells). Notice, that the above **DFS** would always visit these two nodes in a specific order, independent of the structure of the rest of the quadtree. Thus, if $\square$ gets visited before $\widehat{\square}$, we denote this fact by $\square \prec \widehat{\square}$. This defines a total ordering over all canonical squares. It would be in fact useful to extend this ordering to also includes points. Thus, consider a point p and a canonical square $\square$. If $p \in \square$ then we will say that $\square \prec p$. Otherwise, if $\square \in \mathsf{G}_i$, let $\widehat{\square}$ be the cell in G_i that contains p. We have that $\square \prec p$ if and only if $\square \prec \widehat{\square}$. Next, consider two points p and q, and let G_i be a grid fine enough such that p and q lie in two different cells, say, $\square_p$ and $\square_q$, respectively. Then $p \prec q$ if and only if $\square_p \prec \square_q$.

We will refer to the ordering induced by $\prec$ as the $\mathcal{Q}$***-order***.



The ordering $\prec$ when restricted only to points, is the ordering along a space filling mapping that is induced by the quadtree **DFS**. This ordering is know as the $\mathcal{Z}$***-order*** . Note, however, that since we allow comparing cells to cells, and cells to points, the $\mathcal{Q}$-order no longer has this exact interpretation. Furthermore, unlike the Peano or Hilbert curve, our mapping is not continuous. Our mapping has the advantage of being easy to define. Indeed, given a real number $\alpha \in [0, 1)$, with the binary expansion $\alpha = 0.x_1x_2x_3\ldots$ (i.e., $\alpha = \sum_{i=1}^{\infty} x_i 2^{-i}$), our mapping will map it to the point $(0.x_2x_4x_6\ldots, 0.x_1x_3x_5\ldots)$.

2.4.1.1 Computing the $\mathcal{Q}$-order quickly

For our algorithmic applications, we need to be able to find the ordering according to $\prec$ between any two given cells/points quickly. To this end, let $\mathcal{LCA}(p, q)$ of two points $p, q \in [0, 1]^2$ denote the smallest canonical square that contains both p and q. To compute this, let $\mathrm{bit}_\Delta(\alpha, \beta)$ of two

real numbers $\alpha, \beta \in [0,1)$ be the index of the first bit after the period in which they differ. Thus, $\text{bit}_\Delta(1/4, 3/4) = 1$ and $\text{bit}_\Delta(7/8, 3/4) = 3$. Clearly, the level ℓ of $\square = \mathcal{LCA}(p,q)$ is equal to

$$\ell = \min(\text{bit}_\Delta(x_p, x_q), \text{bit}_\Delta(y_p, y_q)) - 1,$$

where x_p and y_p denote the x and y coordinates of p, respectively. Thus, the side length of $\square = \mathcal{LCA}(p,q)$ is $\Delta = 2^{-\ell}$. Let $x' = \lfloor x/\Delta \rfloor$ and $y' = \lfloor y/\Delta \rfloor$. Thus,

$$\mathcal{LCA}(p,q) = [x', x' + \Delta) \times [y', y' + \Delta).$$

The $\mathcal{LCA}$ of two cells is just the $\mathcal{LCA}$ of their centers.

Now, given two cells $\square$ and $\widehat{\square}$, we would like to determine their $\mathcal{Q}$-order. If $\square \subseteq \widehat{\square}$ then $\widehat{\square} \prec \square$. If $\widehat{\square} \subseteq \square$ then $\square \prec \widehat{\square}$. Otherwise, let $\widetilde{\square} = \mathcal{LCA}(\square, \widehat{\square})$. We can now determine which children of $\widetilde{\square}$ contains these two cells, and since we know the traversal ordering among children of a node in a quadtree we can now resolve this query in constant time.

Corollary 2.4.1 *Assuming that the* bit_Δ *operation and the* $\lfloor \cdot \rfloor$ *operation can be performed in constant time, then one can compute* $\mathcal{LCA}$ *of two points (or cells) in constant time. Similarly, the* $\mathcal{Q}$*-order can be resolved in constant time.*

Computing bit_Δ efficiently. It seems somewhat suspicious that one assumes that the bit_Δ operations can be done in constant time on a classical RAM machine. However it is a reasonable assumption on a real world computer. Indeed, in floating point representation, once you are given a number it is easy to access its mantissa and exponent in constant time. If the exponents are different then bit_Δ can be computed in constant time. Otherwise, we can easily *xor* the mantissas of both numbers, and compute the most significant bit that is on. This can be done in constant time by converting the xored mantissa into floating point number, and computing its $\log_2$ (some CPUs have this command built in). Observe, that all these operations are implemented in hardware in the CPU and require only constant time.

2.4.2 Performing a point-location in a quadtree

Let $\mathcal{T}$ be a given quadtree and a query point $\mathsf{q} \in [0,1]^2$. We would like to find the leaf v of $\mathcal{T}$ such that its cell contains q. We assume that $\mathcal{T}$ is given to us as a list of cells stored in an ordered-set data-structure, using the $\mathcal{Q}$-order over the cells.

To answer the query, we first find the two consecutive cells in this list such that $\square \prec \mathsf{q} \prec \widehat{\square}$. It is now easy to verify that $\square$ must be the quadtree leaf containing q. Indeed, let $\square_\mathsf{q}$ be the leaf of $\mathcal{T}$ that its cell contains q. By definition, we have that $\square_\mathsf{q} \prec \mathsf{q}$. Thus, the only bad scenario is that $\square_\mathsf{q} \prec \square \prec \mathsf{q}$. But this implies, by the definition of $\mathcal{Q}$-order, that $\square$ must be contained inside $\square_\mathsf{q}$ contradicting our assumption that $\square_\mathsf{q}$ is a leaf of the quadtree.

Lemma 2.4.2 *Given a quadtree* $\mathcal{T}$ *of size* n*, with its leaves stored in an ordered-set data-structure* $\mathcal{D}$ *according to the* $\mathcal{Q}$*-order, then one can perform point-location query in* $O(Q(n))$ *time, where* $Q(n)$ *is the time to perform a search query in* $\mathcal{D}$*.*

2.4.3 Overlaying two quadtrees

Given two quadtrees $\mathcal{T}'$ and $\mathcal{T}''$ we would like to overlay them to compute their combined quadtree. This is the minimal quadtree such that every cell of either $\mathcal{T}'$ or $\mathcal{T}''$ appears in it. Observe that if the two quadtrees are given as sorted lists of their cells (ordered by the $\mathcal{Q}$-order) then their overlay is just the merged list, with replication removed.

Lemma 2.4.3 *Given two quadtrees $\mathcal{T}'$ and $\mathcal{T}''$ given as sorted lists of their nodes, one can compute the merged quadtree in linear time (in their size) by merging the two sorted lists and removing duplicates.*

2.4.4 Point location in a compressed quadtree

so let $\mathcal{T}$ be a compressed quadtree, that its nodes are stored in an ordered-set data-structure. Let q be the query point. The required node v has $\mathsf{q} \in \mathsf{rg}_v$ and is either a leaf of the quadtree or a compressed node.

If the binary search using the $\mathcal{Q}$-order return a node v, such that $\Box_v \prec \mathsf{q}$ then if $\mathsf{q} \in \Box_v$ then we are done, as v is the required answer. So, if $\mathsf{q} \notin \Box_v$ then it must be that the node u such that $\mathsf{q} \in \mathsf{rg}_u$ is a compressed node. As such, consider the cell $\Box = \mathcal{LCA}(\Box_v, \mathsf{q})$. Clearly, the compressed node w that its region contains q have the property that $\Box \subseteq \Box_w$. Furthermore, let z be the only child of w. We have that $\Box_z \subseteq \Box \subseteq \Box_w$. In particular, in the ordering of nodes by the $\mathcal{Q}$-order we have that $\Box_w \prec \Box \prec \Box_z$, where $\Box_w$ and $\Box_z$ are consecutive in the ordering of the nodes of the compressed quadtree. It follows, that we can find $\Box_w$ by doing an additional binary search for $\Box$ in the ordered set of the nodes of the compressed quadtree. We summarize:

Lemma 2.4.4 *Given a compressed quadtree $\mathcal{T}$ of size n, with its leaves stored in an ordered-set data-structure $\mathcal{D}$ according to the $\mathcal{Q}$-order, then one can perform point-location query in $\mathcal{T}$ in $O(Q(n))$ time, where $Q(n)$ is the time to perform a search query in $\mathcal{D}$.*

2.4.5 Inserting/deleting a point into/from a compressed quadtree

Let q be a point to be inserted into the quadtree, and let w be the node of the compressed quadtree such that $\mathsf{q} \in \mathsf{rg}_w$. There are several possibilities:

- The node w is a leaf, and there is no point associated with it. Then we just stored p at w, and we are done.

- The node w is a leaf, and there is a point p already stored in w. In this case, let $\Box = \mathcal{LCA}(\mathsf{p}, \mathsf{q})$, and insert $\Box$ into the compressed quadtree. Furthermore, split $\Box$ into its children, and also insert the children into the compressed quadtree. Finally, associate p with the new leaf that contains it, and associate q with the leaf that contains it. Note, that because of the insertion w becomes a compressed node if $\Box_w \neq \Box$, and it becomes a regular internal node otherwise.

- The node w is a compressed node. Let z be the child of w, and consider $\Box = \mathcal{LCA}(\Box_z, \mathsf{q})$. Insert $\Box$ into the compressed quadtree if $\Box \neq \Box_w$ (note that in this case w would still be a compressed node, but with a larger "hole"). Also insert all the children of $\Box$ into the

quadtree, and store p in the appropriate child. Hang $\square_z$ from the appropriate child, and turn this child into a compressed node.

In all three cases, the insertion requires a constant number of search/insert operations on the ordered-set data-structure.

Deletion is done in a similar fashion. We delete the point from the node that contains it, and then we trim away nodes that are no longer necessary.

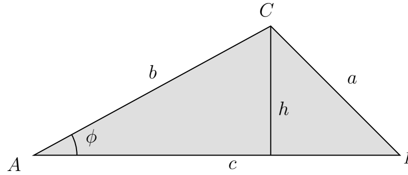
Theorem 2.4.5 *Assuming one can compute the $\mathcal{Q}$-order in constant time, then one can maintain a compressed quadtree of point-set in $O(\log n)$ time per operation, where insertion, deletion and point-location queries are supported. Furthermore, this can implemented using an ordered-set data-structure.*

2.5 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 $\triangle = abc$, the aspect ratio $\mathcal{A}_{\text{ratio}}(\triangle)$ is the length of the longest side divided by the height of the triangle on the longest edge.

Lemma 2.5.1 *Let ϕ be the smallest angle for a triangle. We have that $1/\sin\phi \leq \mathcal{A}_{\text{ratio}}(\triangle) \leq 2/\sin\phi$.*

Proof: Consider the triangle $\triangle = \triangle abc$.



We have $\mathcal{A}_{\text{ratio}}(\triangle) = c/h$. However, $h = b\sin\phi$, 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, $\mathcal{A}_{\text{ratio}}(\triangle) \geq b/h = b/(b\sin\phi) = 1/\sin\phi$. And similarly, $\mathcal{A}_{\text{ratio}}(\triangle) \leq 2b/h = 2b/(b\sin\phi) = 2/\sin\phi$. ■

Another natural measure of sharpness is the ***edge ratio*** $E_{\text{ratio}}(\triangle)$, which is the ratio between a triangle's longest and shortest edges. Clearly, $\mathcal{A}_{\text{ratio}}(\triangle) > E_{\text{ratio}}(\triangle)$, for any triangle $\triangle$. For a triangulation $\mathcal{M}$, we denote by $\mathcal{A}_{\text{ratio}}(\mathcal{M})$ the maximum aspect ratio of a triangle in $\mathcal{M}$. Similarly, $E_{\text{ratio}}(\mathcal{M})$ denotes the maximum edge ratio of a triangle in $\mathcal{M}$.

Definition 2.5.2 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 2.5.3 *Let P be a set of points in the plane, such that* $\mathrm{diam}(P) = \Omega(1)$ *and* $\Phi = \Phi(P)$. *Then, one can compute a (minimal size) balanced quadtree $\mathcal{T}$ of P, in time $O(n\log n + m)$ time, where m is the size of the output quadtree.*

Proof: Compute a compressed quadtree $\mathcal{T}$ of P in $O(n \log n)$ time. Next, we traverse $\mathcal{T}$, and replace every compressed edge of $\mathcal{T}$ by the sequence of quadtree nodes that defines it. To guarantee the balance condition, we create a queue of the nodes of $\mathcal{T}$, and store the nodes of $\mathcal{T}$ 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 $\square_v$ are balanced. Specifically, let c be one of $\square_v$'s neighboring cells in the grid of $\square_v$, and let c_p be the square containing c in a grid one level up. We compute $\mathrm{id}(c), \mathrm{id}(c_p)$, and check if there is a node in $\mathcal{T}$ with those IDs. If not, we create a node w with region c_p and $\mathrm{id}(c_p)$, 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 2.5.4 The ***extended cluster*** of a cell c in a quadtree $\mathcal{T}$ is the set of 5×5 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 $\mathcal{T}$ 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 $\mathcal{T}$ (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 2.5.5 *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 $\mathcal{T}$ of P. Next, for every leaf node v of $\mathcal{T}$ which contains a point of P, we verify that all its extended cluster are leaves of $\mathcal{T}$. 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 $\mathcal{T}$ 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 $\mathcal{T}$ into a triangulation of P.

We "warp" the quadtree framework as follows. Let y be the corner nearest x of the leaf of $\mathcal{T}$ 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 2.2 shows a triangulation resulting from a variant of this method.

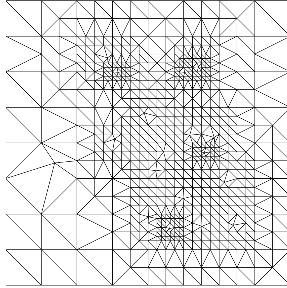


Figure 2.2: A well balanced triangulation.

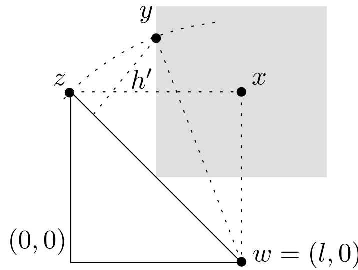


Figure 2.3: Illustration of the proof of Lemma 2.5.6.

Lemma 2.5.6 *The method above gives a triangulation $\mathcal{QT}(P)$ with $\mathcal{A}_{\text{ratio}}(\mathcal{QT}(P)) \leq 4$.*

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 $\mathcal{A}_{\text{ratio}}(\triangle) \leq h/(2a/h) = h^2/2a = 4$.

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 2.3. Using the notation of Figure 2.3, we have $y = (l/2, \sqrt{7}l/2)$. Thus,

$$\mu = \text{area}(\triangle wyz) = \frac{1}{2}\begin{vmatrix} 1 & 0 & l \\ 1 & l & 0 \\ 1 & l/2 & (\sqrt{7}/2)l \end{vmatrix} = \frac{1}{2}\begin{vmatrix} l & -l \\ l/2 & (\sqrt{7}/2-1)l \end{vmatrix} = \frac{\sqrt{7}-1}{4}l^2.$$

We have $h\sqrt{2}l/2 = \mu$, and thus $h' = \sqrt{2}\mu/l = \frac{\sqrt{7}-1}{2\sqrt{2}}l$. The longest distance y can be from w is $\alpha = \sqrt{(1/2)^2 + (3/2)^2}l = (\sqrt{10}/2)l$. Thus, the aspect ratio of the new triangle is bounded by $\alpha/h' = \left(\sqrt{10}/2\right)/\frac{\sqrt{7}-1}{2\sqrt{2}} \approx 2.717 \leq 4$. ■

For a triangulation $\mathcal{M}$, let $|\mathcal{M}|$ denote the number of triangles of $\mathcal{M}$. 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 2.5.7 *There is a constant c', independent of P, such that $|\mathcal{QT}(P)| \leq c' \sum_{\triangle \in \mathcal{DT}(P)} \log E_{\text{ratio}}(\triangle)$.*

Proof: For this lemma, we modify the description of our algorithm for computing $\mathcal{QT}(P)$. We compute the compressed quadtree $\mathcal{T}''$ of P, and we uncompress the edges by inserting missing cells. Next, we split a leaf of $\mathcal{T}''$ if it has side length $\varkappa$, it is not empty (i.e., it contains a point of P), and there is another point of P of distance $\leq 2\varkappa$ from it. We refer to such a node as being ***crowded***. We repeat this, till there are no crowded leaves. Let $\mathcal{T}'$ denote the resulting quadtree. We now iterate over all the nodes v of $\mathcal{T}'$, and insert all the nodes of the extended cluster of v into $\mathcal{T}'$. Let $\mathcal{T}$ denote the resulting quadtree. It is easy to verify that $\mathcal{T}$ is well-balanced, and identical to the quadtree generated by the algorithm of Lemma 2.5.5 (although it is unclear how to implement the algorithm described here efficiently).

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

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 $2\varkappa$ from it, which in the next level is in a cell non-adjacent to the children of c, where $\varkappa$ 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 $2\varkappa$. 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.[2]

[2]To 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 then 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.

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 $2\varkappa$ units apart. These steps are at most $O(\log E_{\text{ratio}}(\triangle y_1 y_2 z))$ quadtree levels apart, so we can charge all the crowded cells caused by Y to $\triangle y_1 y_2 z$. 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 $O(\log E_{\text{ratio}}(\triangle abc))$ for some Delaunay triangles $\triangle abc$. ■

Theorem 2.5.8 *Given any point set* P*, we can find a triangulation* $\mathcal{QT}(P)$ *such that each point of* P *is a vertex of* $\mathcal{QT}(P)$ *and* $\mathcal{A}_{\text{ratio}}(\mathcal{QT}(P)) \leq 4$*. There is a constant* c''*, independent of* P*, such that if* $\mathcal{M}$ *is any triangulation containing the points of* P *as vertices,* $|\mathcal{QT}(P)| \leq c'' |\mathcal{M}| \log \mathcal{A}_{\text{ratio}}(\mathcal{M})$.

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

Proof: Let Y be the set of vertices of $\mathcal{M}$. Lemma 2.5.7 states that there is a constant c such that $|\mathcal{QT}(Y)| \leq c \sum_{\triangle \in \mathcal{DT}(Y)} \log E_{\text{ratio}}(\triangle)$. The Delaunay triangulation has the property that it maximizes the minimum angle of the triangulation, among all triangulations of the point set [For97].

If $Y = P$, then using this maxminŋangle property, we have $\mathcal{A}_{\text{ratio}}(\mathcal{M}) \geq \frac{1}{2}\mathcal{A}_{\text{ratio}}(\mathcal{DT}(P)) \geq \frac{1}{2}E_{\text{ratio}}(\mathcal{DT}(P))$, by Lemma 2.5.1. Hence

$$|\mathcal{QT}(P)| \leq c \sum_{\triangle \in \mathcal{DT}(P)} \log E_{\text{ratio}}(\mathcal{DT}(P)) = c\,|\mathcal{M}|\, E_{\text{ratio}}(\mathcal{DT}(P)) \leq 2c\,|\mathcal{M}|\, \mathcal{A}_{\text{ratio}}(\mathcal{M}).$$

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

Corollary 2.5.9 $|\mathcal{QT}(P)| = O(n log \mathcal{A}_{\text{ratio}}(\mathcal{DT}(P)))$.

Corollary 2.5.9 is tight, as can be easily verified.

2.6 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 [WVTP97]), among a lot of other applications.

The algorithm described, in Section 2.2.1, for the efficient construction of compressed quadtrees is new, as far as I know. 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).

The elegant skip-quadtree is from the recent work of Eppstein *et al.* [EGS05].

The idea of storing a quadtree in an ordered set by using the $\mathcal{Q}$-order on the nodes (or even only on the leaves) is due to Gargantini [Gar82], and it is referred to as ***linear quadtrees*** in the literature. The idea was used repeatedly for getting good performance in practice from quadtrees.

It is maybe beneficial to emphasize that if one does not require the internal nodes of the compressed quadtree for the application, then one can avoid storing them in the data-structure. In fact, if one is only interested in the point themselves, then can even skip storing the leaves themselves, and then the compressed quadtree just becomes a data-structure that stores the points according to their $\mathbb{Z}$-order. This approach can be used for example to construct a data-structure for approximate nearest neighbor [Cha02] (however, this data-structure is still inferior, in practice, to the more optimized but more complicated data-structure of Arya *et al.* [AMN+98]). The author finds that thinking about such data-structures as compressed quadtrees (with the whole additional unnecessary information) more intuitive, but the reader might disagree[3].

$\mathbb{Z}$-order and space filling curves. The idea of using $\mathbb{Z}$-order for speeding up spatial data-structures can be traced back to the above work of Gargantini [Gar82], and it is widely used in databases and seems to improve performance in practice [KF93]. The $\mathbb{Z}$-order can be viewed as a mapping from the unit interval to the unit-square, by splitting the odd bits, of a real number $\alpha \in [0, 1)$, to be the x-coordinate and the even bits of α to encode the y-coordinate of the mapped point. While this mapping is simple to define it is not continuous. Somewhat surprisingly one can find a continuous mapping that maps the unit interval to the unit-square, see Exercise 2.7.4. A large family of such mappings is known by now, see Sagan [Sag94] for an accessible book on the topic.

But is it really practical? Quadtrees seems to be widely used in practice and perform quite well. Compressed quadtrees seems to be less widely used, but they have the benefit of being much simpler than their relatives which seems to be more practical but theoretically equivalent.

Good triangulations. 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 [Üng04]. Furthermore, by merging the quadtree approach with Üngör technique, one can get an optimal running time algorithm [HÜ05].

[3] The author reserves the right to disagree with himself on this topic in the future if the need arise.

2.7 Exercises

Exercise 2.7.1 (Quadtree for fat quadtrees.) [5 Points]

A triangle $\triangle$ is called α-fat if each one of its angles is at least α, where $\alpha > 0$ is a prespecified constant (for example, α 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 2.7.2 (Quadtree construction is tight.) [5 Points]

Prove that the bounds of Lemma 2.2.2 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 $\Omega(n \log \Phi(P))$.

Exercise 2.7.3 (Cell queries.) [10 Points]

Let $\widehat{\square}$ be a canonical grid cell. Given a compressed quadtree $\widehat{T}$, we would like to find the *single* node $v \in \widehat{T}$, such that $P \cap \widehat{\square} = \mathsf{P}_v$. We will refer to such query as a ***cell query***. Show how to support cell queries in compressed quadtree in logarithmic time per query.

Exercise 2.7.4 (Space filling curve.) [10 Points]

The ***Peano curve*** $\sigma : [0, 1) \to [0, 1)^2$, maps a number $\alpha = 0.t_1 t_2 t_3 \ldots$ (the expansion is in base 3) to the point $\sigma(\alpha) = (0.x_1 x_2 x_3 \ldots,\ 0.y_1 y_2 \ldots)$, where $x_1 = t_1$, $x_i = \phi(t_{2i-1}, t_2 + t_4 + \ldots + t_{2i-2})$, for $i \geq 1$. Here, $\phi(a, b) = a$ if b is even and $\phi(a, b) = 2 - a$ if b is odd. Similarly, $y_i = \phi(t_{2i}, t_1 + t_3 + \cdots + t_{2i-1})$, for $i \geq 1$.

(A) **[2 Points]** Prove that the mapping σ covers all the points in the open square $[0, 1)^2$, and it is one to one.

(B) **[8 Points]** Prove that σ is continuous.

Acknowledgments

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

2.8 From previous lectures

Lemma 2.8.1 *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* $\mathrm{radius}(D) \leq 2 r_{\mathrm{opt}}(P, k)$.

Lemma 2.8.2 *For any point set* P*, and* $r > 0$*, we have: (i) For any real number* $A > 0$*, it holds* $\mathtt{depth}(P, Ar) \leq (A+1)^2 \,\mathtt{depth}(P, r)$*, (ii)* $\mathtt{gd}_r(P) \leq \mathtt{depth}(P, r) \leq 9\mathtt{gd}_r(P)$*, (iii) if* $r_{\mathrm{opt}}(P, k) \leq r \leq 2 r_{\mathrm{opt}}(P, k)$ *then* $\mathtt{gd}_r(P) \leq 5k$*, and (iv) Any circle of radius* r *is covered by at least one grid cluster in* G_r*.*

Exercise 2.8.3 (Tail inequality for geometric variables.) [10 Points]

Let $X_1, \ldots, X_m$ be m independent random variables with geometric distribution with probability p (i.e., $\mathbf{Pr}[X_i = j] = (1-p)^{j-1}p$). Let $Y = \sum_i X_i$, and let $\mu = \mathbf{E}[Y] = m/p$. Prove that

$$\mathbf{Pr}[Y \geq (1+\delta)\mu] \leq \exp\left(-\frac{m\delta^2}{8}\right).$$

Bibliography

[AMN+98] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. Assoc. Comput. Mach.*, 45(6), 1998.

[BEG94] M. Bern, D. Eppstein, and J. Gilbert. Provably good mesh generation. *J. Comput. Syst. Sci.*, 48:384–409, 1994.

[Cha02] T. M. Chan. Closest-point problems simplified on the ram. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pages 472–473. Society for Industrial and Applied Mathematics, 2002.

[CK95] P. B. Callahan and S. R. Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *J. Assoc. Comput. Mach.*, 42:67–90, 1995.

[EGS05] D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *Proc. 21st Annu. ACM Sympos. Comput. Geom.*, pages 296–305. ACM, June 2005.

[For97] S. Fortune. Voronoi diagrams and Delaunay triangulations. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 20. CRC Press LLC, Boca Raton, FL, 1997.

[Gar82] I. Gargantini. An effective way to represent quadtrees. *Commun. ACM*, 25(12):905–910, 1982.

[HÜ05] S. Har-Peled and A. Üngör. A time-optimal delaunay refinement algorithm in two dimensions. In *Proc. 21st Annu. ACM Sympos. Comput. Geom.*, pages 228–236, 2005.

[KF93] I. Kamel and C. Faloutsos. On packing r-trees. In *Proc. 2nd Intl. CConf. Info. Knowl. Mang.*, pages 490–499, 1993.

[Mil04] G. L. Miller. A time efficient Delaunay refinement algorithm. In *Proc. 15th ACM-SIAM Sympos. Discrete Algorithms*, pages 400–409, 2004.

[Rup93] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 83–92, 1993.

[Sag94] H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.

[Sam89] H. Samet. *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*. Addison-Wesley, Reading, MA, 1989.

[Üng04] A. Üngör. Off-centers: A new type of steiner points for computing size-optimal quality-guaranteed delaunay triangulations. In *Latin Amer. Theo. Inf. Symp.*, pages 152–161, 2004.

[WVTP97] M. Waldvogel, G. Varghese, J. Turener, and B. Plattner. Scalable high speed ip routing lookups. In *Proc. ACM SIGCOMM 97*, Octeber 1997.