Dynamic Smooth Compressed Quadtrees

Ivor Hoog v.d.[1]
Dept. of Inform. and Computing Sciences, Utrecht University, the Netherlands
i.d.vanderhoog@uu.nl

Elena Khramtcova[2]
Computer Science Department, Université libre de Bruxelles (ULB), Belgium
elena.khramtsova@gmail.com

Maarten Löffler[3]
Dept. of Inform. and Computing Sciences, Utrecht University, the Netherlands
m.loffler@uu.nl

Abstract

We introduce dynamic smooth (a.k.a. balanced) compressed quadtrees with worst-case constant time updates in constant dimensions. We distinguish two versions of the problem. First, we show that quadtrees as a space-division data structure can be made smooth and dynamic subject to *split* and *merge* operations on the quadtree cells. Second, we show that quadtrees used to store a set of points in $\mathbb{R}^d$ can be made smooth and dynamic subject to *insertions* and *deletions* of points. The second version uses the first but must additionally deal with *compression* and *alignment* of quadtree components. In both cases our updates take $2^{\mathcal{O}(d \log d)}$ time, except for the point location part in the second version which has a lower bound of $\Omega(\log n)$; but if a pointer (*finger*) to the correct quadtree cell is given, the rest of the updates take worst-case constant time. Our result implies that several classic and recent results (ranging from ray tracing to planar point location) in computational geometry which use quadtrees can deal with arbitrary point sets on a real RAM pointer machine.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis

Keywords and phrases smooth, dynamic, data structure, quadtree, compression, alignment, Real Ram

Digital Object Identifier 10.4230/LIPIcs.SoCG.2018.45

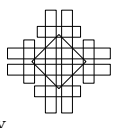
Related Version A full version is avaliable at [14], `https://arxiv.org/abs/1712.05591`.

Acknowledgements The authors would like to thank Joe Simons and Darren Strash for their inspiring initial discussion of the problem.

1 Introduction

The quadtree is a hierarchical spacial subdivision data structure based on the following scheme: starting with a single square, iteratively pick a square and subdivide it into four equal-size smaller squares, until a desired criterion is reached. Quadtrees and their higher-dimensional

[1] Supported by the Netherlands Organisation for Scientific Research (NWO) through project no 614.001.504.
[2] Supported by the SNF Early Postdoc Mobility grant P2TIP2-168563, Switzerland, and F.R.S.-FNRS, Belgium.
[3] Partially supported by the Netherlands Organisation for Scientific Research (NWO) through project no 614.001.504



equivalents have been long studied in computational geometry [26, 7, 17, 5, 11, 6, 18, 3] and are popular among practitioners because of their ease of implementation and good performance in many practical applications [12, 20, 4, 1, 9, 2]. We will not review the extremely rich literature here and instead refer to the excellent books by Samet [24] and Har-Peled [13].

Smooth and dynamic quadtrees

A quadtree is *smooth*[4] if each leaf is comparable in size to adjacent leaves. It has been long recognized that smooth quadtrees are useful in many applications [4], and smooth quadtrees can be computed in linear time (and have linear complexity) from their non-smooth counterparts [8, Theorem 14.4].

A quadtree is *dynamic* if it supports making changes to the structure in sublinear time. Recently, quadtrees have been applied in kinetic and/or uncertain settings that call for dynamic behaviour of the decomposition [9, 19, 15]. Bennett and Yap [3] show how to maintain a smooth *and* dynamic quadtree subject to amortized constant-time *split* and *merge* operations on the quadtree leaves.

At this point, it is useful to distinguish between the quadtree *an sich*, a combinatorial subdivision of space, and the quadtree as a data structure for storing a point set (or other set of geometric objects). Given a set P of points in the plane and a square that contains them, we can define the minimal quadtree that contains P to be the quadtree we obtain by recursively subdividing the root square until no leaf contains more than one (or a constant number of) point(s). It is well-known that such a minimal quadtree can have superlinear complexity, but can still be stored in linear space by using *compression* [8]. Additionally, when working in the Real RAM computation model, it may not be possible to keep different compressed components properly aligned [13, 18]. These complications imply that we cannot simply apply results known for standard/regular (henceforth called *uncompressed*) quadtrees. When maintaining a dynamic quadtree storing a point set P, we wish to support high-level operations of inserting points into P and removing points from P.[5]

Contribution

In this paper, we show that it is possible to maintain a quadtree storing a set of points P that is smooth and possibly compressed, which supports worst-case constant-time insertions and deletions of points into P, assuming we are given a pointer (*finger*) to the cell of the current quadtree containing the operation. Our result runs on a Real RAM pointer machine and generalises to arbitrary constant dimensions.

In the first half of the paper (Sections 3-5), we focus on the problem of making the quadtree itself dynamic and smooth, improving the recent result by Bennett and Yap [3] from amortized to worst-case constant time split and merge operations. The challenge here is to avoid cascading chains of updates required to maintain smoothness. Our key idea is to introduce several layers of smoothness: we maintain a core quadtree which is required to be 2-smooth, but cells added to satisfy this condition themselves need only be 4-smooth, etc (refer to Section 2 for the formal definition of 2^j-smoothness). In Section 3, we show that when defining layers in this way, we actually need only two layers in $\mathbb{R}^2$, and the second

[4] Also called *balanced* by some authors, which is not to be confused with the notion of balance in trees related to the relative weights of subtrees.

[5] In Table 1 in Section 2 we provide a complete list of operations and how they relate.

layer will always already be smooth. In Section 4, we show that we can handle updates on the core quadtree in constant time. In Section 5, we generalise the result to arbitrary dimensions (now, the number of layers, and thus the smoothness of the final tree, depends on the dimension).

In the second half of the paper (Sections 6-7), we focus on lifting our result to quadtrees that store a set of points on a pure real-valued pointer machine. The challenge here is to redefine compressed quadtrees in a consistent way across different layers of smoothness, and to re-align possibly misaligned components on the fly when such components threaten to merge. In Section 2, we show that we can view insertion of a point as a two-step procedure, where we first need to locate the correct leaf of the current tree containing the new point, and then actually insert it into the quadtree. We show in Section 6 that we can still handle the second step in worst-case constant time. In Section 7, we deal with the issue of avoiding the use of the floor operation, which is not available on a pure Real RAM.

Implications

Many publications in computational geometry use a concept which we shall dub **principal neighbor access**: The idea that for any cell C we can find its relevant neighbors in constant time:

▶ **Definition 1.** Given a quadtree T over $\mathbb{R}^d$, we say that we have **principal neighbor access** if for any cell C in T we can find the smallest cells C' in T with $|C'| \geq |C|$ and C' neighboring C in constant time if d is constant.

Bennet and Yap in [3] implement principal neighbor access by storing explicit **principle neighbor pointers** to the larger neighbors C'. Khramtcova and Löffler in [15] achieve principal neighbor access with the well known **level pointers** on a smooth quadtree. These two unique ways to guarantee principle neighbor access were also noted by Unnikrishnan *et al.* in [25] where a *threaded* quadtree in [25] maintained the equivalent to **principle neighbor pointers** as opposed to a *roped* quadtree in [23] which only maintained level pointers.

Principle neighborhood access allows us to traverse the neighborhood of any cell C in constant time. Bennet and Yap observe that any (non-compressed) quadtree must be smooth to dynamically maintain level pointers by using a sequence of cells that they insert. In the full version [14] we show that if quadtrees are compressed, you even need $\Theta(n)$ time for a single split operation to update the level pointers. For this reason Bennet and Yap develop their amortized-constant dynamic smooth quadtrees in $\mathbb{R}^d$ in [3]. We note that most applications that use principal neighbor access (dynamic variant of collision detection [21], ray tracing [1, 20] and planar point location [15, 19]) often run many operations parallelized on the GPU. In such an environment amortized analysis can become troublesome since there is a high probability that at least one GPU-thread obtains the worst-case $\mathcal{O}(n)$ running time. In that scenario the other threads have to wait for the slow thread to finish so the computations effectively run in $\mathcal{O}(n)$ time which makes our worst-case constant time algorithm a vast improvement.

A large number of papers in the literature explicitly or implicitly rely on the ability to efficiently navigate a quadtree, and our results readily imply improved bounds from amortized to worst-case [20, 15, 19, 9, 22], and extends results from bounded-spread point sets to arbitrary point sets [10]. Other papers could be extended to work for dynamic input with our dynamic quadtree implementation [1, 18, 21]. Several dynamic applications are in graphics-related fields and are trivially parallelizable, which enhanced the need for worst-case

bounds. In the full version [14] we give a comprehensive overview of the implications of our result.

2 Preliminaries

In this section we review several necessary definitions. Concepts that were already existing prior to this work are underlined. Consider the d-dimensional real space $\mathbb{R}^d$. For a hypercube $R \subset \mathbb{R}^d$, the **size** of R, denoted $|R|$, is the length of a 1-dimensional facet (i.e., an edge) of R.

▶ **Definition 2** (Quadtree)**.** Let R be an axis-aligned hypercube in $\mathbb{R}^d$. A **quadtree** T on the root cell R is a hierarchical decomposition of R into smaller axis-aligned hypercubes called **quadtree cells**. Each node v of T has an associated quadtree cell C_v, and v is either a leaf or it has 2^d equal-sized children whose cells subdivide C_v.[6]

From now on, unless explicitly stated otherwise, when talking about a quadtree cell C we will be meaning both the hypercube C and the quadtree node corresponding to C.

▶ **Definition 3** (Neighbor, Sibling neighbor)**.** Let C and C' be two cells of a quadtree T in $\mathbb{R}^d$. We call C and C' **neighbors**, if they are interior-disjoint and share (part of) a $(d-1)$-dimensional facet. We call C and C' **sibling neighbors** if they are neighbors and they have the same parent cell.

▶ Observation 1. Let C be a quadtree cell. Then: (i) C has at most $2d$ neighbors of size $|C|$; and (ii) For each of the d dimensions, C has exactly one sibling neighbor that neighbors C in that dimension.

▶ **Definition 4** (2^j-smooth cell, 2^j-smooth quadtree)**.** For an integer constant j, we call a cell C 2^j**-smooth** if the size of each leaf neighboring C is at most $2^j|C|$. If every cell in a quadtree is 2^j-smooth, the quadtree is called 2^j**-smooth**.

▶ Observation 2. If all the quadtree leaves are 2^j-smooth, then all the intermediate cells are 2^j-smooth as well.[7] That is, the quadtree is 2^j-smooth.

▶ **Definition 5** (Family related)**.** Let C_1, C_2 be two cells in a quadtree T such that $|C_1| \leq |C_2|$. If the parent of C_2 is an ancestor of C_1 we call C_1 and C_2 **family related**.[8]

We now consider quadtrees that store point sets. Given a set P of points in $\mathbb{R}^d$, and a hypercube R containing all points in P, an **uncompressed quadtree** that stores P is a quadtree T in $\mathbb{R}^d$ on the root cell R, that can be obtained by starting from R and successively subdividing every cell that contains at least two points in P into 2^d child cells.

▶ **Definition 6** (Compression)**.** Given a large constant α, an α-compressed quadtree is a quadtree with additional **compressed nodes**. A compressed node C_a has only one child C with $|C| \leq |C_a|/\alpha$, and the region $C_a \setminus C \subset \mathbb{R}^d$ does not contain any points in P. We call the link between C_a and C a **compressed link**, and C_a the **parent** of the compressed link.

Compressed nodes induce a partition of a compressed quadtree T into a collection of uncompressed quadtrees interconnected by compressed links. We call the members of such a collection the **uncompressed components** of T.

---

[6] We follow [18] in using *quadtree* in any dimension rather than dimension-specific terms (i.e. *octree*, etc).
[7] Observe that if a single (leaf) cell is 2^j-smooth, its ancestors do not necessarily have to be such.
[8] Observe that C_1 and C_2 do not have to be neighbors.

Table 1 Operations considered in this paper and the running times of the provided implementation.

Operation	Running time
I. Quadtree operations (uncompressed quadtree)	
Split a cell	$\mathcal{O}((2d)^d)$
Merge cells	$\mathcal{O}((2d)^d)$
II. Quadtree operations (α-compressed quadtree)	
Insert a component	$\mathcal{O}(d^2(6d)^d)$
Delete a component	$\mathcal{O}(d^2(6d)^d)$
Upgrowing of a component	$\mathcal{O}(\log(\alpha)d^2(6d)^d)$
Downgrowing of a component	$\mathcal{O}(\log(\alpha)d^2(6d)^d)$
III. Operations on the point set P, stored in a quadtree	
Insert a point into P	$\mathcal{O}(d\log(n) + \log(\alpha)d^2(6d)^d)$
Insert a point into P, given a finger	$\mathcal{O}(\log(\alpha)d^2(6d)^d)$
Delete a point from P	$\mathcal{O}(\log(\alpha)d^2(6d)^d)$

2.1 Quadtree operations and queries

Table 1 gives an overview of quadtree operations. It is insightful to distinguish three levels of operations. Operations on a compressed quadtree (II) internally perform operations on an uncompressed quadtree (I). Similarly, operations on a point set stored in a quadtree (III) perform operations on the compressed quadtree (II). We now give the formal definitions and more details.

▶ **Definition 7** (split, merge)**.** Given a leaf cell C of a quadtree T, the **split** operation for C inserts the 2^d equal-sized children of C into T. Given a set 2^d leaves of a quadtree T which share a parent, the **merge** operation removes these 2^d cells. The parent cell is now a leaf in T.

▶ **Definition 8** (upgrowing, downgrowing)**.** Let A be an uncompressed component of a compressed quadtree with a root R. **Upgrowing** of A adds the parent R' of R to T. The cell R' becomes the root of component A. **Downgrowing** of A removes the root R of A, and all the children of R except one child C. Cell C becomes the root of A. The downgrowing operation requires R to be an internal cell and all the points stored in A to be contained in one child C of R.

An insertion of a point p into the set P stored in an α-compressed quadtree T is performed in two phases: first, the leaf cell of T should be found that contains p; second, the quadtree should be updated. The first phase, called **point location**, can be performed in $\mathcal{O}(d\log(n))$ time (see edge oracle trees in [19]), and can be considered a query in our data structure. We refer to the second phase separately as **inserting a point given a finger**, see Table 1.

3 Static non-compressed smooth quadtrees in $\mathbb{R}^1$ and $\mathbb{R}^2$

We first view the quadtree as a standalone data structure subject only to merge and split operations. In this section we are given a unique non-smooth, uncompressed quadtree T_1 over $\mathbb{R}^1$ or $\mathbb{R}^2$ with n cells. It is known [8, Theorem 14.4] that an uncompressed quadtree can be made smooth by adding $\mathcal{O}(n)$ cells. However, the reader can imagine that if we want all the cells to be 2-smooth that we cannot make the quadtree dynamic with worst-case constant updates because balancing keeps cascading. In this section we show that if T_1 is

a quadtree over $\mathbb{R}^1$ or $\mathbb{R}^2$ then we can extend T_1 by consecutively adding $d \in \{1, 2\}$ sets of cells,[9] i. e., cells of d different *brands*, so that in the resulting *extended* quadtree T^* each cell is smooth according to its brand. The total number of added cells is $2^{\mathcal{O}(d \log(d))}n$.

3.1 Defining our smooth quadtree

We want to add a minimal number of cells to the original quadtree T_1 such that the cells of T_1 become 2-smooth and the balancing cells are smooth with a constant dependent on $d \in \{1, 2\}$. In general we want to create an **extended quadtree** T^* with $T_1 \subset T^*$ where all cells with brand j are 2^j-smooth for $j \leq d + 1$.

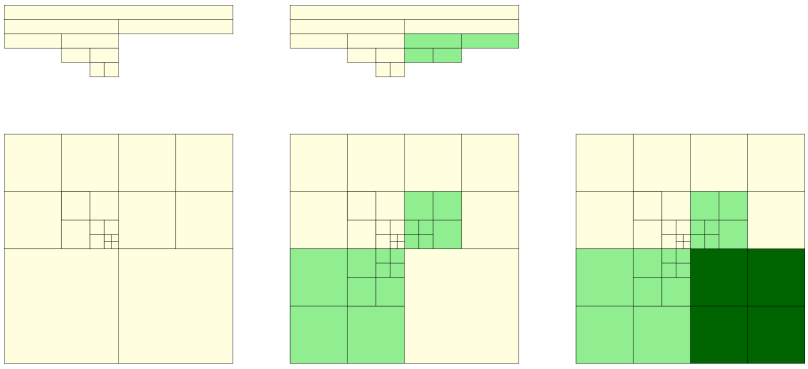


Figure 1 Left: a quadtree in $\mathbb{R}^1$ (up) and $\mathbb{R}^2$ (down); Center: the (light-green) cells of brand 2 added; Right: the (dark-green) cells of brand 3 added. In each row, the rightmost tree is the smooth version of the leftmost one.

The *true* cells (T_1) get brand 1. Figure 1 shows two quadtrees and its balancing cells. This example also illustrates our main result: to balance a tree T_1 over $\mathbb{R}^d$ we use $(d + 1)$ different brands of cells and the cells of the highest brand are automatically 2^{d+1}-smooth. This example gives rise to an intuitive, recursive definition for balancing cells in $\mathbb{R}^d$. In this definition we have a slight abuse of notation: For each brand j, we let T_j denote the set of cells with brand j, and T^j denote the quadtree associated with the cells in the sets T_i for all $i \leq j$:

▶ **Definition 9** (sets T_j)**.** Let T_1 be a set of true cells in $\mathbb{R}^d$. We define the sets $T_j, 2 \leq j \leq d+1$ recursively:

Given a set of cells T_j in $\mathbb{R}^d$, let T^j be the quadtree corresponding to $\cup_{i \leq j} T_i$.

We define the set T_{j+1} to be the minimal set of cells obtained by splitting cells of T^j, such that each cell in T_j is 2^j-smooth in T_{j+1}.

For each set T_j, to every cell in T_j we assign brand j.

▶ **Definition 10.** In $\mathbb{R}^d$, we define T^* to be T^{d+1}.

The extended quadtree T^* has three useful properties which we prove in the remainder of this section: the tree is unique, the size of the tree is linear in n, and cells in the tree which are related in ancestry have a related brand.

▶ **Lemma 11.** *For a given set T_j, the set T_{j+1} is unique.*

[9] In Section 5 we show the same is possible for quadtrees of arbitrary dimension d.

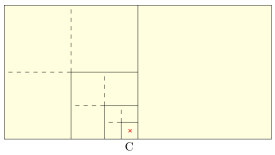


Figure 2 Let the figure show $T_{j-1} = T_1$, the true cells of a quadtree in $\mathbb{R}^2$ in white and denote the cell with the red cross as C. Cells shown with dotted lines exist but are not important for the example. Note that C is adjacent to a cell of 8 times its size so C is not 2-smooth and the parent of C is also not 2-smooth. If we want to split the neighbor of C into cells with brand 2 we create cells which are 4 times the size of C and we thus balance its parent. The second split creates cells of brand 2 which are twice the size of C and so C is 2-smooth.

Proof. Per definition a cell C is 2^j-smooth if all its neighboring leaf cells are at most a factor 2^j larger than C. This means that if we want to balance a cell C then we need to check for each of its neighboring cells if it is too large and if so, add a minimum number of cells accordingly. This makes the minimum set of cells that balances a cell C unique. If for each cell in T_j, its balancing cells are unique, then the set T_{j+1} (the union of all sets of balancing cells) is unique. ◀

▶ **Lemma 12.** *Every quadtree T^j has less than $\mathcal{O}((d2^d)^j n)$ leaf cells.*

Proof. We prove this by induction. By definition T_1 has $\mathcal{O}(n)$ cells and all cells in T_j exist to balance cells in T_{j-1}.

Now we assume that the tree T^{j-1} defined for T_1 has $\mathcal{O}((d2^d)^{j-1}n)$ leaf cells. Each of these leaf cells C can have at most d leaf neighbors which are larger than C. If such a cell C is not 2^{j-1}-smooth we need to split the too large neighboring cells. Observe that for each split we either fix the balance of C or the balance of an ancestor of C which was also in T_{j-1} and which also had to be 2^{j-1}-smooth (see Figure 2).

The result of this observation is that we perform at most d splits per leaf in T^{j-1}. Each split creates 2^d cells so T^j must have at most $d2^d\mathcal{O}((d2^d)^{j-1}n) = \mathcal{O}((d2^d)^j n)$ leaf cells.[10] ◀

▶ **Corollary 13.** *If d is constant, the tree T^* has size $\mathcal{O}(n)$.*

▶ **Lemma 14.** *Let C_1, C_2 be two family related (possibly non-leaf) cells in T^* such that $|C_1| \leq |C_2|$. Then the brand of C_2 is at most the brand of C_1.*

Proof. The proof is a proof per construction where we try to reconstruct the sequence of operations that led to the creation of cell C_1. All of the ancestors of C_1 must have a brand lower or equal to the brand of C_1, this includes the parent C_a of C_2. Since C_1 is a descendant of C_a, C_a must be split. In that split all of the children of C_a (including C_2) are created with a brand lower or equal to the brand of C_1. ◀

▶ **Lemma 15** (The Branding Principle)**.** *Let C_j be a cell in T^* with brand j. Then for any cell N neighbouring C with $|N| \geq 2^j|C_j|$, the brand of N is at most $j + 1$.*

Proof. This property follows from the definition of each set of cells T_j. If C_j is 2^j-smooth, its neighboring cells N can be at most a factor 2^j larger than C_j. When we define T_{j+1}, all

[10] We improve this bound to $\mathcal{O}(d^j 2^d n)$ leaf cells in the full version by observing that for every balance split, there are at least $\frac{1}{2}2^d$ cells demanding that the same cell must be split.

the neighbors of C_j either already have size at most $2^j|C_j|$ and thus a brand of at most j, or the neighbors must get split until they have size exactly $2^j|C_j|$. When the latter happens those cells get brand $j+1$. ◀

With these lemmas in place we are ready to prove the main result for static uncompressed quadtrees in $\mathbb{R}^1$ and $\mathbb{R}^2$.

3.2 Static uncompressed smooth quadtrees over $\mathbb{R}^1$

Let T_1 be a non-compressed quadtree over $\mathbb{R}^1$ which takes $\mathcal{O}(n)$ space. In this subsection we show that we can add at most $\mathcal{O}(n)$ cells to the quadtree T_1 such that all the cells in the resulting quadtree are 2^j-smooth for some $j \leq 2$ and the true cells are 2-smooth. Lemma 12 tells us that we can add at most $\mathcal{O}(n)$ cells with brand 2 to T_1 resulting in the tree $T^* = T_1 \cup T_2$ where all the true cells are 2-smooth in T^*.

Our claim is that in a static non-compressed quadtree over $\mathbb{R}^1$ all the cells in T_2 must be 4-smooth in T^* since we cannot have two neighboring leaf cells in T_2 with one cell more than a factor 2 larger than the other.

▶ **Theorem 16.** *Let T_1 be an uncompressed quadtree over $\mathbb{R}^1$ which takes $\mathcal{O}(n)$ space. In the smooth tree T^* there cannot be two neighboring leaf cells C_2, C_3, both with brand 2, such that $|C_2| \leq \frac{1}{2^2}|C_3|$.*[11]

Proof. The proof is by contradiction and is illustrated in Figure 3. Assume for the sake of contradiction that we have two neighboring cells C_2 and C_3 both with brand 2 with $|C_3| = 4|C_2|$. C_2 has two neighbors: one family related neighbor and one non-sibling neighbor. C_3 cannot be contained in a sibling neighbor because C_3 is larger than C_2. Note that C_2 exists to balance a true cell C_1 of smaller size. C_1 cannot be a descendant of C_3 because C_3 is a cell with brand 2. So C_1 must be a descendant of the sibling neighbor of C_2. That would make C_2 and C_1 family related and Lemma 14 then demands that C_2 has brand at most 1; a contradiction. ◀

3.3 Static uncompressed smooth quadtrees over $\mathbb{R}^2$

We also show that we can make a smooth non-compressed static quadtree over $\mathbb{R}^2$ that takes $\mathcal{O}(n)$ space, such that all the cells in the quadtree are 2^j-smooth for a $j \leq 3$. We denote the original cells by T_1 and we want them to be 2-smooth. We claim that in the extended quadtree T^* (as defined in Definition 10) all cells are 8-smooth.

[11] Careful readers can observe two things in this section: (i) Cells which are 2-smooth are allowed to have neighbors which are 4 times as large but in $\mathbb{R}^1$ they cannot. (ii) The proof of this theorem actually shows that C_3 can not even be a factor two larger than C_2. We choose not to tighten the bounds because these two observations do not generalize to higher dimensions.

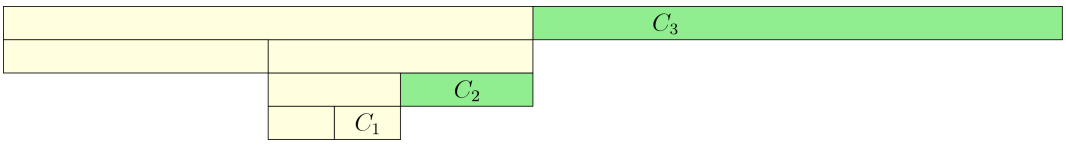


Figure 3 Two neighboring cells with brand 2 in a one-dimensional quadtree. In the figure white cells have brand 1 and light green cells have brand 2.

▶ **Theorem 17.** *Let T_1 be an uncompressed quadtree over $\mathbb{R}^2$ which takes $\mathcal{O}(n)$ space. In the extended tree T^* there cannot be two neighboring leaf cells C_3, C_4, both with brand 3, such that $|C_3| \leq \frac{1}{2^3}|C_4|$.*

Proof. The proof resembles the proof of Theorem 16, and it is illustrated in Figure 4. However, it requires two cases instead of one. Note that for C_3 to exist there must be at least two consecutive neighbors of C_3, (C_2 and C_1) with brand 2 and 1 respectively, such that $|C_1| = \frac{1}{2}|C_2| = \frac{1}{2} \cdot \frac{1}{4}|C_3|$.

Observe that C_1, C_2 and C_3 cannot be family related because of Lemma 14 and observe that C_4 can not be a sibling neighbor of C_3. The proof claims that it is impossible to place C_1, C_2, C_3 and C_4 in the plane without either violating the branding principle (Lemma 15), Lemma 14, or causing a cell with brand 1 or 2 to be not smooth.

Our first claim is that C_3 must share a vertex with C_4 (and similarly, C_2 must share a vertex with C_3). If this is not the case all the neighbors of C_3 (apart from C_4) are either contained in sibling neighbors of C_3 or neighbors of C_4. However that would imply that either C_2 is family related to C_3 or that an ancestor of C_2 of size $|C_3|$ is a neighbor of C_4. The first case cannot happen because of Lemma 14, in the second case we have a cell with brand 2 neighboring C_4 which is $\frac{1}{8}$'th the size of C_4 so C_4 must have been split but C_4 must be a leaf. Without loss of generality we say that C_3 shares the top left vertex with C_4 (Figure 4).

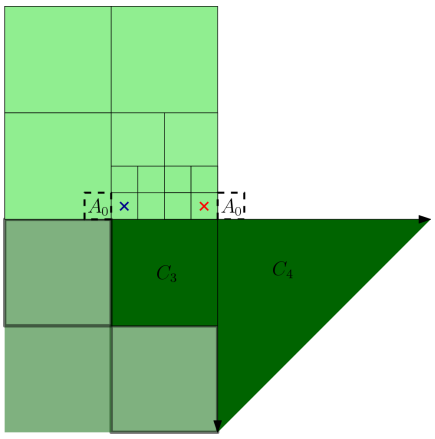


Figure 4 Two neighboring cells with brand 3 in a two-dimensional quadtree. The cells with brand 2 are light green and cells with brand 3 are dark green.

Since C_2 cannot be placed in a sibling neighbor of C_3, C_2 must be placed in the positive $\vec{y}$ direction from C_3. C_2 must also share a vertex with C_3 so we distinguish between two cases: C_2 shares the top left vertex with C_3 or the top right.

Case 1: The top left vertex. In this case C_2 is the blue square in Figure 4. C_1 cannot be contained in a sibling neighbor of C_2 so C_1 must lie to the left. However if C_1 is adjacent to C_2, its parent A_0 (the dashed lines in the figure) must also be a neighbor of C_2. Because we placed C_4 and C_3 without loss of generality, Figure 4 shows us that A_0 must neighbor a sibling neighbor of C_3 which we will denote as $F(C_3)$. We know that $|F(C_3)| = |C_3| \geq 2^1|A_0|$ and that A_0 has brand 1 and $F(C_3)$ has brand 3. This is a contradiction with the *Branding Principle* (Lemma 15).

Case 2: The top right vertex. In this case C_2 is the red square in the figure. C_1 cannot be contained in a sibling neighbor of C_2 so C_1 must lie to the right. However, if C_1 is adjacent to C_2, its parent A_0 (the dashed lines in the figure) must also be a neighbor of C_2. Moreover (for similar reasons as the first case) A_0 must also be a neighbor of C_4. We know that A_0 is the ancestor of a true cell, so A_0 has brand 1. Moreover, $|C_4| \geq 8|A_0|$ so C_4 must have been split which contradicts that C_4 is a leaf.

Both cases lead to a contradiction so Theorem 17 is proven. The structure of this proof is identical to the structure of the proof of the generalized theorem in the full version. ◀

4 Dynamic quadtrees

In the previous section we have shown that, given a static uncompressed quadtree T_1 of $\mathcal{O}(n)$ size over $\mathbb{R}^1$ or $\mathbb{R}^2$, we can create a static smooth tree T^* of $\mathcal{O}(n)$ size. In this section we prove that if T_1 is a dynamic tree, we can also dynamically maintain its extended variant T^*.

Let T_1 be a quadtree over $\mathbb{R}^1$ or $\mathbb{R}^2$ subject to the split and merge operation (Table 1 I.). If we use the split operation to create new true cells in T_1 then in T^* we (possibly) need to add cells (to the set T_2) that smooth the new true cells. Similarly if we add cells to T_2 we might need to add cells to T_3. The first question that we ask is: can we create a new split and merge operation that takes a constant number of steps per operation to maintain T^*?

▶ **Lemma 18.** *Given an uncompressed non-smooth quadtree T_1 in $\mathbb{R}^d$ of $\mathcal{O}(n)$ size and its extended tree T^*. Let T_1' be an uncompressed non-smooth quadtree such that T_1 can become T_1' with one merge or split operation. Then T^* and T'^* differ by at most $(2d)^d$ quadtree cells.*

Proof. In the proof of Lemma 12 we showed that each cell in T_j had at most d balancing cells in T_{j+1}. So if we add a new cell in T_1 with the split operation, we need to do at most d^d split operations to create the d^d cells to smooth the tree up to the level $(d+1)$.[12] d^d split operations create at most $(2d)^d$ cells.

Lemma 11 states that for each set of true cells T_1 the tree T^* is unique. So if we want to **merge** four cells in T_1 we must get a new unique T_1' and T'^*. We know that we can go from T'^* to T^* with d^d **split operations**, so we can also go from T^* to T'^* with d^d **merge** operations. ◀

4.1 The algorithm that maintains T^*

Lemma 18 tells us that it should be possible to dynamically maintain our extended quadtree T^* with $2^{\mathcal{O}(d\log(d))}$ operations per split or merge in the true tree T_1. The lemma does not specify which cells exactly need to be split. We note that our extended quadtree T^* is unique and thus independent from the order in which we split cells in T_1. In the full version we show that this property prohibits the naive implementation (just split all cells which conflict with another cell's smoothness): this does not maintain a quadtree that follows the definition of T^* given by Definition 9 (and thus does not have to be smooth). Instead we introduce the following lemma which will help us design a correct algorithm for maintaining T^*:

▶ **Lemma 19.** *Given a dynamic quadtree T_1 and its dynamic extended quadtree T^* with $T_1 \subset T^*$ where T^* is defined according to Definition 9. If a cell $C \in T^*$ has brand $j > 1$ then there is at least one neighbor N of C such that N has brand $j-1$ and $|C| = 2^{j-1}|N|$.*

Proof. If C has brand j, then according to Definition 9, C exists to smooth a cell $N \in T_{j-1}$. Per definition N has brand $j-1$ so $|C|$ is indeed $2^{j-1}|N|$. ◀

This observation allows us to devise an algorithm that maintains T^* after a split operation in T_1, we call this algorithm the **aftersplit procedure**. The first change to our static construction is that each cell $C \in T^*$ gets a collection of brands (which can contain duplicates). The current brand of a cell is the minimum of its collection of brands. Given this new definition of a brand we define a two-phased procedure:

[12] Section 5 will show that the $(d+1)$'th level is always 2^{d+1}-smooth

- Whenever we split a cell C with brand j, we check all the d neighboring *leaf* cells N that are larger than C. If N is more than a factor 2^{j-1} larger than C, the new children of C are non-smooth and N should be split into cells with brand $j+1$. If a neighbor N is split, we also invoke the aftersplit procedure on N.
- Secondly we consider this: for any neighboring leaf N of C we check there exists a cell C' with C' equal to N or an ancestor of N with the following property: C' is exactly a factor 2^{j-1} larger than C. In that case C' could exist to smooth the new children of C so we add the brand $(j+1)$ to its set of brands. We call this **rebranding**.

Algorithm 1 The procedure for after splitting a cell.

```
1: procedure AfterSplit(Cell C, Integer j)
2:     for Cell N ∈ LargerLeafNeighbors do
3:         if |N|/|C| > 2^{j-1} then
4:             V ← Split(N, j+1)
5:             AfterSplit(N, j+1)
6:         if ∃C' ∈ N ∪ Ancestors(N) such that |C'|/|C| = 2^{j-1} then
7:             C'.Brands.Add(j+1)
8:             if Changed(C'.Brands.Minimum) then
9:                 AfterSplit(C', C'.Brands.Minimum)
```

▶ **Lemma 20.** *If we split a cell with brand j, the aftersplit procedure performs at most $(2d)^{d-j}$ split and merge operations and the resulting extended quadtree T^* implements Definition 9.*

Proof. Observe that if we split a cell C with brand j, C has at most d neighbors which are at least a factor 2^{j-1} larger than C and a leaf in T^*. We can find the larger leaf neighbors with at most d level pointer traversals and the neighbors of exactly 2^{j-1} size by first finding an ancestor of C and using that ancestor's level pointers. If we **rebrand** or split one of the found neighbors, that neighbor gets a brand one greater than j so the aftersplit procedure will recurse with a new $j' = j+1$. This means that we recurse at most $d-j$ times which makes the aftersplit procedure on a cell with brand j perform d^{d-j} split operations which creates at most $(2d)^{d-j}$ cells.

The resulting tree T^* must implement Definition 9 because of Lemma 19. This lemma states that any neighboring cell N with $|N| = 2^{j-1}|C|$ could exist to smooth a child cell of C in the static scenario, so each N should contain the option to have that brand. ◀

The AfterMerge procedure is simply the inverse of the Aftersplit procedure. If we merge cells into a cell C, we check all neighbors of size $2^{j-1}|C|$ and $2^{j-2}|C|$. In the first case, we remove the brand j from the cell and check if it still has to exist. In the second case, we remove the brand j from the cell and check if it needs to be rebranded to a higher brand.

▶ **Theorem 21.** *For each dynamic compressed quadtree T_1 over $\mathbb{R}^d$ we can maintain the extended variant T^* with at most $\mathcal{O}((2d)^d)$ split and merge operations in T^* per one split or merge operation in T_1.*

5 Extending the proof of Section 3 to work for $\mathbb{R}^d$

In this section, we prove that T^* is smooth in every dimension d if T_1 is an uncompressed quadtree. Due to space restrictions, we only summarize the result here; the details can be found in the full version of the paper.

In Section 4 we already elaborated on how we can dynamically maintain our extended quadtree T^* in at most $\mathcal{O}((2d)^d)$ operations per split and merge on the true tree T_1. What remains to be proven is that the extended quadtree is indeed a smooth quadtree. We prove this by proving a generalized version of Theorem 17 in Section 3:

▶ **Theorem 22.** *Let T_1 be non-compressed quadtree over $\mathbb{R}^d$ which takes $\mathcal{O}(n)$ space. In the extended tree T^* there cannot be two neighboring leaf cells which we will name C_{d+1}, C_{d+2} with both brand $(d+1)$ such that $|C_{d+1}| \leq \frac{1}{2^{d+1}}|C_{d+2}|$.*

The proof is similar to the proof in Section 3.3. If the two neighboring cells C_{d+1} and C_{d+2} want to exist in the plane then there must be a **chain** of d cells (C_d to C_1) with decreasing brand that forces the cells to exist. We show that it is impossible to embed this chain into the plane without either needing to split the largest cell or violating the *Branding Principle* (Lemma 15). The difficulty of this proof is that there are many ways to embed such a chain of cells in $\mathbb{R}^d$. Moreover, if you want a cell C_j in the chain to cause a contradiction with a neighboring cell you need to prove that the neighbor actually exists. We define an abstract virtual operator that can traverse the extended quadtree. With that operator and Lemma 14 in place we show that if you want to place C_{d+2} to C_1 in the pane, then each time we must let C_j neighbor C_{j+1} in a unique dimension. If we use a dimension $\vec{y}$ twice we distinguish between two cases:

The first case is that we use the same dimension and same direction. This case is equal to case 1 in Section 3.3 and we show that we must violate the *Branding Principle*.

The second case is that we use the same dimension and opposite direction and it corresponds with case 2 in Section 3.3. In this case we show that C_{d+2} must have been split.

The result is that we need $(d+1)$ different dimensions to embed the chain but we have only d, a contradiction.

6 Compressed quadtrees in $\mathbb{R}^d$

The remainder of our work focuses on building smooth quadtrees that store a set of points and that are dynamic with respect to insertions and deletions of points from the set. If we want to build a dynamic quadtree over a point set of unbounded spread we need the quadtree to be compressed. We again want to define an extended quadtree T^* that we can maintain with the operations in Section 4. In this section summarize the extention of our results to compressed quadtrees. The details can be found in the full version of the paper.

Recall that in Definition 10 we had $(d+1)$ different sets of cells where each set T_j was defined as the minimal number of cells needed to balance the cells in T_{j-1}. We showed that if we have two uncompressed quadtrees T_1 and T_1' which only differ by one split/merge operation that their extended quadtrees T^* and T'^* only differ in $\mathcal{O}((2d)^d)$ cells. However, if T_1 and T_1' are uncompressed quadtrees and if we use Definition 9, the reader can imagine that there are scenarios where their extended quadtrees T^* and T'^* differ in an arbitrarily large number of cells (see the full version). This is why instead we redefine our extended quadtree as the union of all extended quadtrees A^* of all uncompressed components A_1 of T_1: $T^* := \cup_{Uncompressed\ components A_1} A^*$. If we only demand that cells in uncompressed components count towards smoothness (the other cells are not stored anyway) then this extended quadtree is clearly smooth if Theorem 22 holds. However, Theorem 22 relies on Lemma 14 that says that cells that are *family related* have a related brand and with this definition of T^* we can place cells with an arbitrary high brand "on top" of other cells. We observe that the proof of Theorem 22 only uses paths that traverse d^2 levels of depth in the

tree and we define additional operators that make sure that for each cell C, all ancestors within d^2 levels of C have a correct brand. The result is the following theorem:

▶ **Theorem 23.** *For each dynamic compressed quadtree T_1 over $\mathbb{R}^d$ we can maintain a smooth variant T^* with at most:*

- $\mathcal{O}((2d)^d)$ *operations per split or merge on T_1.*
- $\mathcal{O}(\log(\alpha)d^2(6d)^{d+1})$ *operations per deletion or insertion of a compressed leaf.*
- $\mathcal{O}(d^2(6d)^d)$ *time per upgrowing or downgrowing.*

7 Alignment in Real RAM

Finally, we show that we can maintain non-aligned compressed quadtrees. Here we only give a summary of this part; for the details see the full version.

In the previous section we were treating our compressed quadtree as if for each compressed component A_1, the cell of its root R was *aligned* with the cell of the leaf C_a that stores its compressed link. That is, the cell of R can be obtained by repeatedly subdividing the cell of C_a. However, finding an aligned cell for the root of a new uncompressed component when it is inserted is not supported in constant time in Real RAM model of computation. The reason is that the length of the compressed link, and thus the number of divisions necessary to compute the aligned root cell, may be arbitrary.

Instead of computing the aligned root cell at the insertion of each new compressed component, we allow compressed nodes to be associated with any hypercube of an appropriate size that is contained in the cell of the compressed ancestor C_a. While doing so, we ensure that the following **alignment property** is maintained: For any compressed link in T^*, if the length of the link is at most 4α, then the corresponding uncompressed component is aligned with the parent cell of the link.

In the full version of [19], Löffler *et al.* obtain this in amortized additional $O(1)$ time by relocating the uncompressed component just before the decompression operation starts. To accomplish the relocation task, they exploit the algorithm of Löffler and Mulzer [18] that, given a compressed quadtree T in $\mathbb{R}^2$ and an appropriate square S, produces a 2-smooth compressed quadtree T' on S that stores the same point set as T. We modify the result of [19] in three ways: (i) We de-amortize it by adapting a well-known de-amortization technique that gradually constructs the correctly aligned copy of an unaligned uncompressed component [16]. (ii) We adapt the algorithm of [18] so that it produces our target quadtree T^* instead of a 2-smooth compressed quadtree. (iii) By extending the analysis in [18], we show that the algorithm to produce T^* works in $\mathbb{R}^d$, and its time complexity is linear in the size of the relocated tree.

The main result of this section is as follows.

▶ **Theorem 24.** *Let P be a point set in $\mathbb{R}^d$ and T_1 be an α-compressed quadtree over P for a sufficiently big constant α with $\alpha > 2^{2^d}$, and let T^* be the extended variant of a compressed quadtree T_1 with the definition in [14]. The operations split and upgrowing on T^* can be modified to maintain the alignment property for T^*. The modified operations require worst-case $O(32^d + d^2 \cdot (6d)^d)$ time.*

References

1 Boris Aronov, Hervé Brönnimann, Allen Y Chang, and Yi-Jen Chiang. Cost prediction for ray shooting in octrees. *Computational Geometry*, 34(3):159–181, 2006.

2 Huck Bennett, Evanthia Papadopoulou, and Chee Yap. Planar minimization diagrams via subdivision with applications to anisotropic Voronoi diagrams. *Comput. Graph. Forum*, 35(5):229–247, 2016.

3 Huck Bennett and Chee Yap. Amortized analysis of smooth quadtrees in all dimensions. *Computational Geometry*, 63:20–39, 2017.

4 Marshall Bern, David Eppstein, and John Gilbert. Provably good mesh generation. *Journal on Computational System Sciences*, 48(3):384–409, 1994.

5 Marshall Bern, David Eppstein, and Shang-Hua Teng. Parallel construction of quadtrees and quality triangulations. *International Journal on Computational Geometry and Applications*, 9(6):517–532, 1999.

6 Kevin Buchin and Wolfgang Mulzer. Delaunay triangulations in $O(\texttt{sort}(n))$ time and more. *Journal of the ACM*, 58(2):Art. 6, 2011.

7 Paul B. Callahan and S. Rao Kosaraju. A decomposition of multidimensional point sets with applications to k-nearest-neighbors and n-body potential fields. *Journal of the ACM*, 42(1):67–90, 1995.

8 Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 2008.

9 Mark De Berg, Marcel Roeloffzen, and Bettina Speckmann. Kinetic compressed quadtrees in the black-box model with applications to collision detection for low-density scenes. *Algorithms–ESA 2012*, pages 383–394, 2012.

10 Olivier Devillers, Stefan Meiser, and Monique Teillaud. Fully dynamic delaunay triangulation in logarithmic expected time per operation. *Computational Geometry*, 2(2):55–80, 1992.

11 David Eppstein, Michael Goodrich, and Jonathan Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. *International Journal on Computational Geometry and Applications*, 18(1–2):131–160, 2008.

12 Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inform.*, 4:1–9, 1974.

13 Sariel Har-Peled. *Geometric approximation algorithms*, volume 173. American mathematical society Boston, 2011.

14 Ivor Hoog v.d., Elena Khramtcova, and Maarten Löffler. Dynamic smooth compressed quadtrees. *arXiv preprint arXiv:1712.05591*, 2017.

15 Elena Khramtcova and Maarten Löffler. Dynamic stabbing queries with sub-logarithmic local updates for overlapping intervals. In *Proc. 12th Int. Computer Science Symp. in Russia, (CSR)*, pages 176–190, 2017.

16 S Rao Kosaraju and Mihai Pop. De-amortization of algorithms. In *International Computing and Combinatorics Conference*, pages 4–14. Springer, 1998.

17 Drago Krznaric and Christos Levcopoulos. Computing a threaded quadtree from the Delaunay triangulation in linear time. *Nordic Journal on Computating*, 5(1):1–18, 1998.

18 Maarten Löffler and Wolfgang Mulzer. Triangulating the square and squaring the triangle: quadtrees and Delaunay triangulations are equivalent. *SIAM Journal on Computing*, 41(4):941–974, 2012.

19 Maarten Löffler, Joseph A. Simons, and Darren Strash. Dynamic planar point location with sub-logarithmic local updates. In *13th Int. Symp. Algorithms and Data Structures (WADS)*, pages 499–511. Springer Berlin Heidelberg, 2013. full version: arXiv preprint arXiv:1204.4714.

20 J David MacDonald and Kellogg S Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6(3):153–166, 1990.

21 Johannes Mezger, Stefan Kimmerle, and Olaf Etzmuß. Hierarchical techniques in collision detection for cloth animation. In *Proc. Int. Conf. in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG)*, 2003.

22 Eunhui Park and David Mount. A self-adjusting data structure for multidimensional point sets. *Algorithms–ESA 2012*, pages 778–789, 2012.

23 Hanan Samet. Neighbor finding techniques for images represented by quadtrees. *Computer graphics and image processing*, 18(1):37–57, 1982.

24 Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley, Boston, MA, USA, 1990.

25 A Unnikrishnan, Priti Shankar, and YV Venkatesh. Threaded linear hierarchical quadtrees for computation of geometric properties of binary images. *IEEE transactions on software engineering*, 14(5):659–665, 1988.

26 Pravin M. Vaidya. Minimum spanning trees in k-dimensional space. *SIAM J. Comput.*, 17(3):572–582, 1988. doi:10.1137/0217035.