Dynamic Planar Point Location with Sub-logarithmic Local Updates

Maarten Löffler[1], Joseph A. Simons[2], and Darren Strash[2]

[1] Dept. of Information and Computing Sciences, Utrecht University
[2] Dept. of Computer Science, University of California, Irvine

Abstract. We study planar point location in a collection of disjoint fat regions, and investigate the complexity of *local updates*: replacing any region by a different region that is "similar" to the original region. (i.e., the size differs by at most a constant factor, and distance between the two regions is a constant times that size). We show that it is possible to create a linear size data structure that allows for insertions, deletions, and queries in logarithmic time, and allows for local updates in sub-logarithmic time on a pointer machine. We also give results parameterized by the fatness and similarity of the objects considered.

1 Introduction

Planar point location lies at the heart of many geometric problems, and has been a major research topic in computational geometry for the past 40 years. In the static version of the problem, one aims to store a subdivision of the plane such that given a query point q in the plane, the cell of the subdivision containing q can be retrieved quickly [7,14]. In the dynamic version of the problem, one also allows changes to the data set, typically adding or removing line segments to the subdivision [3,10].

The best known dynamic data structures on a real RAM are due to Cheng and Janardan [5], who achieve $O(\log^2 n)$ queries and $O(\log n)$ updates, and Arge *et al.* [2], who achieve $O(\log n)$ queries, $O(\log^{1+\varepsilon} n)$ insertions, and $O(\log^{2+\varepsilon} n)$ deletions. A central open problem in this area is whether a linear-size data structure exists that can support both queries *and* updates in logarithmic time, although this is known to be possible in more specific settings such as monotone or rectilinear subdivisions [10]. Husfeldt *et al.* [11] prove that even in the very strong *cell probe model*, there are $\Omega(\log n / \log\log n)$ lower bounds on both queries and updates.

Despite these theoretical results, practical evidence suggests that *updating* a data structure should be fast. Intuitively, an update to a data set should not need to depend on n at all, unless we need to find the place where the update takes place (i.e., we need to do a point location query). Realistic input models are intended for designing algorithms that are provably efficient in practice, and the fat-and-disjoint model is ubiquitous (see e.g. [6]). In this paper, we study point location data structures on a collection of disjoint fat objects in the plane that support *local updates*: replace any region by a different region that is *similar* to the original. We show that the lower bounds on updates can be broken in this setting, while still allowing $O(\log n)$ queries and using $O(n)$ storage.

The idea of local updates is not new. For example, Nekrich [13] considers (on a word-RAM) the local update operation $\text{insert}_\Delta(x, y)$ which inserts a new element x into a 1-dimensional sorted list, given a pointer to an existing element y that satisfies $|x - y| \leq \Delta$ for some distance parameter Δ. There is also a related concept called *finger updates*, where the position of the update is known; see e.g. Fleischer [9]. However, our results are the first in this area that work in a geometric setting, and they can be implemented on a real-valued pointer machine.

1.1 Problem Description

We define the problem in general dimension d, but restrict our attention to $d \in \{1, 2\}$ in the remainder of this paper. Throughout this paper, we use $|R|$ to denote the diameter of a region $R \subset \mathbb{R}^d$, that is, $|R| = \max_{p,q \in R} |pq|$. We say two *fat*[1] regions $R_1, R_2 \subset \mathbb{R}^d$ are *ρ-similar* if $|R_1 \cup R_2| \leq \rho \min\{|R_1|, |R_2|\}$, see Figure 1.[2]

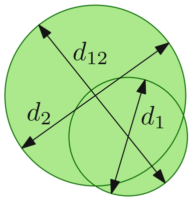

Fig. 1. ρ-similar

Problem 1. Given a set $\mathcal{R}$ of n disjoint fat regions in $\mathbb{R}^d$, store them in a data structure that allows:

– queries: given a point $q \in \mathbb{R}^d$, return the region in $\mathcal{R}$ that contains q (if any) in $Q(n)$ time;
– local updates: given a region $R \in \mathcal{R}$ and a region R' that is ρ-similar to R, replace R by R' in the data structure in $U(n)$ time; and
– global updates: delete an existing region R from the data structure or insert a new region R' into the data structure in $Q(n) + U(n)$ time

such that $Q(n) = O(\log n)$ but $U(n) = o(\log n)$. Note that a local update allows for an arbitrary number of smaller regions to be "between" the old region R and the new region R'.

1.2 Applications

A natural application of our data structure is to keep track of moving objects. One may imagine a number of objects of different sizes moving unpredictably in an environment at different speeds. A popular method for dealing with moving objects is to discretize time and process the new locations of the objects at each time step. The naive way to do this is to simply rebuild an entire data structure

[1] We formally define fat regions in Section 4.

[2] This definition captures two ideas at once: firstly, the *sizes* of R_1 and R_2 can differ by at most a factor of ρ, and secondly, the *distance* between R_1 and R_2 can be at most a factor ρ times the smaller of these sizes.

every time step. Our data structure can be used to process such changes more efficiently.

A different reason for studying this problem comes from the desire to cope with *data imprecision*. One way to model an imprecise point is to keep track of a region of possible locations of the point. Although algorithms to deal with static imprecise data are beginning to be well understood, little effort has been devoted to dealing with *dynamic* imprecise points. However, imprecision is often inherently dynamic (e.g. time-dependent or "stale" data), or explicitly made dynamic (e.g. updates from new samples of the same point). Our data structure can be used to store a set of dynamic imprecise points for quickly answering identity queries (i.e., given a query point, is there a point in the data structure that is potentially equal to the query point).

1.3 Results

We show that given constant similarity and fatness parameters:

– A set of n disjoint intervals in $\mathbb{R}^1$ can be maintained in an $O(n)$ size data structure that supports $O(\log n)$ worst-case time insertion, deletion, and point location queries, and $O(1)$ worst-case time local updates (Section 3).
– A set of n disjoint fat regions in $\mathbb{R}^2$ can be maintained in an $O(n)$ size data structure that supports $O(\log n)$ worst-case time insertion, deletion and point location queries, and $O(\log \log n)$ worst-case time local updates (Section 4).
– We also give bounds that can handle arbitrary similarity and fatness parameters in Theorem 1 and Theorem 2 for the $\mathbb{R}^1$ and $\mathbb{R}^2$ case respectively.

Our data structures can be implemented on a real-valued pointer machine. Because of space restrictions, many proofs and details are omitted. We also refer the interested reader to a full, uncompressed version of this text [12].

2 Tools

Quadtrees. Let B be an axis-aligned square.[3] A *quadtree* T on B is a hierarchical decomposition of B into smaller axis-aligned squares called quadtree *cells*. Each node v of T has an associated cell $C_v \subset \mathbb{R}^d$, and v is either a leaf or has 2^d equal-sized children whose cells subdivide C_v [8]. We denote the parent of a node v by $\bar{v}$. A pair of cells are called *neighbors* if they are interior disjoint and meet at an edge or corner. A leaf v is *α-balanced* if $\alpha|C_v| \geq |C_u|$ for every larger neighbor C_u of C_v. We say T is *α-balanced* if every leaf in T is α-balanced. If α is a small constant, then we simply call the quadtree T *balanced*.

Let $P \subset \mathbb{R}^d$ be a set of n points contained in B. We say T is a *valid* quadtree for P if every leaf of T contains at most 1 point of P. T may have unbounded

[3] We use the term *square* to mean a d-dimensional hypercube, since our main focus is on $d = 2$.

depth if P has unbounded spread,[4] Given a constant a, an *a-compressed* quadtree replaces some paths in T with *compressed* nodes. A compressed node v has only one child $\tilde{v}$ with $|C_{\tilde{v}}| \leq |C_v|/a$ and such that $C_v \setminus C_{\tilde{v}}$ has no points from P.[5] We assume for convenience that $\tilde{v}$ is *aligned* with v, i.e. if we keep subdividing C_v we will eventually create $C_{\tilde{v}}$.[6]

The compressed nodes of a quadtree T cut the tree into a number of components that correspond to smaller regular (uncompressed) quadtrees. We say T is α-balanced if all these smaller trees are α-balanced. It follows directly from Bern *et al.* [4], that a balanced compressed quadtree of linear complexity exists for any set of points P.

Static edge-oracle trees. Let T be an abstract tree with constant maximum degree d. Suppose that the nodes of T are given unique labels, and suppose that each edge $e \in T$ has an oracle which for any node label x can answer the following question: "If we removed e such that T is split into two components, which component would contain the node labeled x?" The edge-oracle tree is a search structure built over the edges of T which allows us to navigate from any node $u \in T$ to any other node $v \in T$ in $O(\log |T|)$ time and examines only $O(\log |T|)$ edges. We can construct an edge-oracle tree for T by recursively locating an edge which divides T into two components of approximately equal size.

Local updates. For a one-dimensional ordered list, data structures that can handle local (finger) updates are well known. One of the simplest implementations on a pointer machine is due to Fleischer [9].

Marked-ancestor problem. Suppose we have a tree in which nodes may be marked or unmarked. Given a node x, we want to answer the query, "Which is the lowest marked ancestor of x in the tree?". This is known as the *marked-ancestor problem.* We also want to support updates, in which nodes are marked or unmarked, and insertions/deletions of nodes to/from the tree. Alstrup *et al.* [1] gave the following results for the marked-ancestor problem on a word-RAM.

Lemma 1. *We can maintain a data structure over any rooted tree T which supports insertions and deletions of leaves in $O(1)$ amortized time, marking and unmarking nodes in $O(\log\log n)$ worst-case time, and marked ancestor queries in $O(\log n/\log\log n)$ worst-case time.*

[4] The *spread* of a point set P is the ratio between the largest and the smallest distance between any two distinct points in P.

[5] Such nodes are also often called *cluster*-nodes in the literature [4].

[6] While this assumption is realistic in practice, on a pure real-valued pointer machine it is not possible to align compressed nodes of arbitrary size difference in constant time. In the full version [12], we show how to adapt the results to unaligned compressed nodes.

2.1 New Tools

Dynamic balanced quadtrees. A *dynamic* quadtree is a data structure that maintains a quadtree Q on a point set P under insertion and deletion of points. In order to maintain a valid quadtree of linear size, we respond with *split* and *merge* operations respectively. A split operation takes a leaf v of Q and adds 2^d children to it; a merge operation takes 2^d leaves with a common parent and removes them. Details are given in the full version [12].

Lemma 2. *We can maintain 4-balance in a dynamic compressed quadtree in $O(1)$ worst-case time per update.*

Dynamic edge-oracle trees. There have been several recent results which generalize classic one-dimensional dynamic structures to a multidimensional setting by combining classic techniques with a quadtree-style space decomposition. However, surprisingly there are no multidimensional data structures which incorporate finger searching techniques, i.e. structures that are able to support both logarithmic queries and worst-case constant time local updates on a quadtree. We show how to build a dynamic edge-oracle tree which combines tree-decomposition and finger searching techniques with a quadtree to support $O(\log n)$ queries and $O(1)$ local updates. Details are given in the full version [12].

Lemma 3. *Let P be a set of n points, and Q be a balanced and compressed quadtree on P. We can maintain P and Q in a data structure that supports $O(\log n)$ point location queries in Q, and local insertions and deletions of points in P (i.e., when given the corresponding cells of Q) in $O(1)$ time.*

Marked-ancestor trees. We show how to answer marked-ancestor queries on a pointer-machine. Details are given in the full version [12].

Lemma 4. *We can maintain a data structure over any rooted tree T which supports insertions and deletions of leaves in $O(1)$ amortized time, marking and unmarking nodes in $O(\log \log n)$ worst-case time, and queries for the lowest marked ancestor in $O(\log n)$ worst-case time. All operations are supported on a pointer machine.*

3 One-Dimensional Case

Our 1D data structure illustrates the key ideas of our approach while being significantly simpler than the 2D version. Note that in $\mathbb{R}^1$, our input set $\mathcal{R}$ of geometric regions is a set of non-overlapping intervals. The difficulty of the problem comes from the fact that a local update may replace any interval by another interval of similar size at a distance related to that size; hence, it may "jump" over an arbitrary number of smaller intervals. Our solution works on a pure Real-valued pointer machine, and achieves constant time updates.

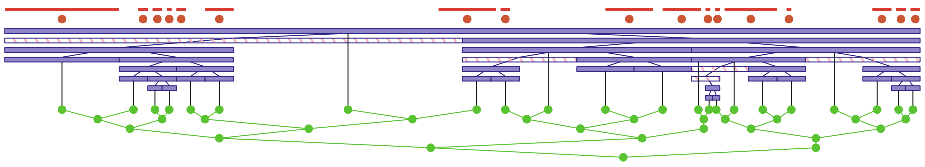

Fig. 2. A set of disjoint intervals and their center points (red); a 4-balanced compressed quadtree on the center points (blue); and a search tree on the leaves (or parts of internal cells not covered by children) of the quadtree (green).

3.1 Definition of the Data Structure

Our data structure consists of two trees. The first is designed to facilitate efficient updates and the second is designed to facilitate efficient queries. The update tree is a compressed quadtree on the center points of the intervals. The quadtree stores a pointer to each interval in the leaf that contains its center point. We also augment the tree with level-links, so that each cell has a pointer to its adjacent cells of the same size (if they exist), and maintain balance in the quadtree as described in Lemma 2. The leaves of the quadtree induce a linear size subdivision of the real line; the query tree is a search tree over this subdivision[7] that allows for fast point location and constant time local updates. We also maintain pointers between the leaves of the two trees, so that when we perform a point location query in the query tree, we also get a pointer to the corresponding cell in the quadtree, and given any leaf in the quadtree, we have a pointer to the corresponding leaf in the query tree. Figure 2 illustrates the data structure.

Details of the following results can be found in the full version [12].

Lemma 5. *Let $I \in \mathcal{R}$ be an interval, and let I' be another interval that is $O(\rho)$-similar to I. Suppose we are given a quadtree storing the midpoints of the intervals in $\mathcal{R}$ and a pointer to the leaf containing the midpoint of I. Then we can find the leaf which contains the midpoint of I' in $O(\log \rho)$ time.*

Theorem 1. *We can maintain a linear size data structure over a set of n non-overlapping intervals such that we can perform point location queries and insertion and deletion of intervals in $O(\log n)$ worst-case time and local updates in $O(\log \rho)$ worst-case time.*

4 Two-Dimensional Case

We now focus our attention on disjoint fat regions in the plane. Intuitively, a fat region should not have any long skinny pieces. We consider two types of fat regions which

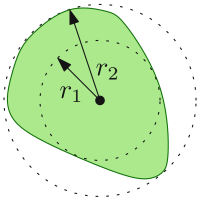

Fig. 3. β-thick

[7] Although we could technically use a search tree directly on the original intervals, we prefer to see it as a tree over the leaves of the quadtree tree in preparation for the situation in $\mathbb{R}^2$.

precisely capture this intuition: *thick* convex regions and *wide* polygons. We say R is β*-thick* if there exists a pair of concentric balls I, O with $I \subseteq R \subseteq O$ and $|O| \leq \beta|I|$, see Figure 3.

Let $\delta \geq 1$. A δ*-corridor* is a isosceles trapezoid whose slanted edges are at most δ times as long as its base. A simple polygon P is δ*-wide* if any isosceles trapezoid $T \subset P$ whose slanted edges lie on the boundary of P is a δ-corridor [15], see Figure 4.[8] Note that any δ-wide polygon R of constant complexity is also β-thick, with $\beta \in \Theta(\delta)$.

We will first solve the problem for convex thick regions, and then extend the result to non-convex wide polygons. Analogously to the 1D case, we will store for each region $R \in \mathcal{R}$ a *representative point* p that lies somehow "in the middle" of R. When the regions are β-thick, we will use the center point of the two concentric disks from the thickness definition as representative point. We denote the set of representative points of the regions in $\mathcal{R}$ by P. Let T be the quadtree built over P. We distinguish between *true* cells, which are necessary in any valid compressed quadtree over P, and B-cells, which may further subdivide a true cell and are only added in order to maintain balance. We store each representative point m in T according to the following rule: Let C_v be the smallest quadtree cell containing m. If C_v is a true cell, then m is stored in v. If C_v is a B-cell, then m is stored in u, the lowest (not necessarily proper) ancestor of v in T such that $|C_u| \geq |R|/(4\beta)$.

Several new problems are introduced which were not present in the 1D case. We briefly sketch how to address each of these problems, and then present the complete solution.

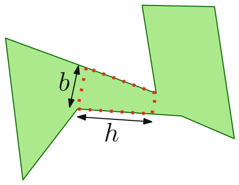

Fig. 4. δ-wide

Linear distance. When performing a query in the one-dimensional case, the location in the quadtree of any intersecting region is at most a constant number of cells away. However, in the two-dimensional case, the location of an intersecting region may be up to a linear number of cells away, as shown in Figure 5(a). We solve this problem with some additional bookkeeping. Given a quadtree cell C_q, we use two different strategies to locate regions intersecting C_q depending on their size. All regions of size at least $2\beta|C_q|$ will be located using a *marked-ancestor* data structure: an additional search structure which we explain in more detail below. All regions of size less than $2\beta|C_q|$ which intersect C_q will register a bidirectional pointer with C_q using the following *tagging* strategy.

Let d be the smallest diameter of a quadtree cell such that $d \geq |R|/(4\beta)$. Let S_R be the set of quadtree cells C which intersect R and are either a leaf or have size $|C| = d$. All cells in S_R will be *tagged* with a pointer to R. Since the quadtree is balanced, given a pointer to any cell in S_R, we can locate all cells in S_R in

[8] Many other notions of fatness exist in the literature. We chose to use thickness because it is basic and implied by most other definitions, and wideness because it will be convenient to use Theorem 3.

$O(|S_R|)$ time. By the following lemma, S_R must contain the cell containing the representative point of R.

Lemma 6. *Let R be a β-thick region stored by our data structure. If C is the quadtree cell which stores the representative point of R, then C has side length at least $\frac{|R|}{4\beta}$.*

Proof. If C is a B-cell, then the claim is true by construction. Suppose C is a true cell. Let m be the representative point of R. By the definition of thickness, there exists a disk $I \subseteq R$ centered at m with $|I| \geq |R|/\beta$. I contains no representative points of regions other than R. Let C be the cell containing m. Note that if C contains m and is significantly smaller than $|R|$, then C must be completely contained in I. However, C must be the largest quadtree cell completely contained in I, since if the parent $\bar{C}$ of C in the quadtree is completely contained in R, then $\bar{C}$ would not have been further subdivided because $\bar{C}$ would contain no other points. Therefore, $\bar{C}$ must have some portion outside of I and must have size larger than $|I|/2$. Thus the size of C is at least $|I|/4 \geq |R|/(4\beta)$. □

Moreover, by the following lemma $|S_R| = O(\beta)$, and therefore, given the cell containing the representative point of R we can tag all cells in S_R in $O(\beta)$ time.

Lemma 7. *Let R be a β-thick region stored in our data structure, and let C be quadtree cell that stores the representative point of R. Then there are at most $O(\beta)$ quadtree cells of size $|C|$ required to cover R.*

Proof. Let I be the largest inscribed disk of R. The boundary of I touches the boundary of R in two or three points. If two points, then these are diametral on I, so R is contained in a strip of width $|I|$. If three points, take the diametral points of these three points and take the strips of width $|I|$ of these three pairs; R is contained in the union of these three strips. Now, if R is beta-thick, the portion of the strips it can be in is at most $\beta|I|$ long. So, R can be covered by $O(\beta)$ disks the size of I. Each such disk can be covered by at most $O(1)$ cells of size $|C|$, by Lemma 6. Thus, $O(\beta)$ cells are required to cover R. □

Linear overlap. In the one-dimensional case, we store only the center points of our regions, and the number of regions that overlap any quadtree cell is at most three. In two dimensions, it appears that we may have a large number of small regions that intersect a quadtree cell. However, we show in the following lemma that this is not the case.

Lemma 8. *The number of β-thick convex regions intersecting any balanced quadtree leaf is $O(\beta)$.*

Proof. Let R_C be the set of thick convex regions that intersect the boundary of leaf C, and let r be the radius of a large disk D containing all regions in R_C. For each region $R_j \in R_C$ there exists a disk $I_j \subseteq R_j$ with center m_j such that $|I_j| \geq |R_j|/\beta$. Moreover, since each region R_j is convex, it must contain a

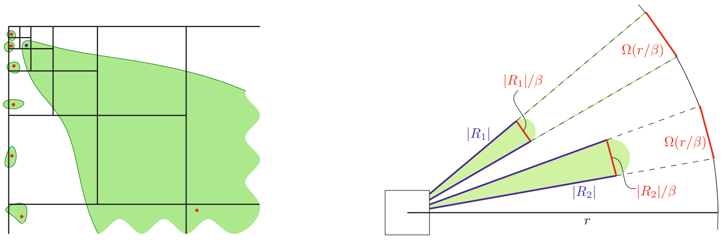

Fig. 5. (a) The intersecting region could be stored a linear distance from the query cell (containing the blue point). (b) The number of regions which can intersect quadtree leaf C is at most $O(\beta)$, since each region blocks a $\Omega(1/\beta)$ fraction of a large circle centered at C, by similar triangles.

triangle consisting of the diameter of I_j and some point $p_j \in R_j \cap C$. Each of the four sides of C can "see" at most πr of the perimeter of D. However, by a similar triangles argument each triangle must block the line of sight from one or more sides to at least $\Theta(r/\beta)$ of the perimeter (see Figure 5(b)). Thus, since the regions are convex and disjoint, the number of regions in R_C is at most $O(\beta)$. □

4.1 Definition of the Data Structure

At the core, our data structure is similar to the one-dimensional data structure described above: we have a spacial tree, which allows for efficient updates, and a search tree, which allows for efficient searching over the quadtree. However, our data structure is augmented to address the problems introduced by the two-dimensional case. We maintain a dynamic balanced quadtree Q over P, which we augment to support *mark* and *unmark* operations and marked-ancestor queries, and we maintain a dynamic edge-oracle tree on the edges of Q.

Marked-ancestor tree. Suppose we are given an angle ϕ which divides 2π (i.e., $k\phi = 2\pi$), and consider the set of angular intervals $\Phi_i = [i\phi, (i+1)\phi]$ (modulo 2π), for integers $1 \leq i \leq k$. For each quadtree cell C of Q with center point c, we define the wedge W_C^i centered at c and with opening angle ϕ to be the union of all halflines from c in a direction in Φ_i. Let $\mathcal{W}_C = \{W_C^i \mid 1 \leq i \leq k\}$; note that $\mathcal{W}_C$ partitions $\mathbb{R}^2$ into k wedges.

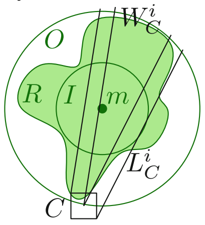

Fig. 6. Illustrating the claim

For each $1 \leq i \leq k$, let T_i be a marked-ancestor structure on Q. We mark a cell C in T_i if and only if there is a region $R \in \mathcal{R}$ of size $2\beta|C| \leq |R| < 4\beta|C|$ that intersects C, and such that the center point of R lies in W_C^i.

When doing a query, we will only look at the first marked ancestor in each T_i. Lemma 9 captures the essential property of the regions which enables this strategy. First, we need the following claim.

Claim. Let β be given and set $\phi = \frac{2\pi}{\lceil 13\beta \rceil}$. Let C be a cell that is marked in T_i by a β-thick region R. Let L_C^i be the set of lines that start in C, and have a direction in Φ_i. Then every line in L_C^i intersects R.

Proof. Let m be the representative point of R. Since R is β-thick, there exist disks $I \subseteq R \subseteq O$ centered at m with $|O|/|I| \leq \beta$. Since R caused C to be marked, O, must intersect C, and m must lie in W_C^i. See Figure 6.

Now, we need that I intersects all lines in L_C^i. The distance from m to C is at most $\frac{1}{2}|O| \leq \frac{\beta}{2}|I|$. Then, the distance from m to the far edge of W_C^i is at most $\frac{\beta}{2}|I|\sin\phi$, and the distance to the far edge of L_C^i is at most $\frac{\beta}{2}|I|\sin\phi + \frac{1}{2}|C|$. Since $|R| \geq 2\beta|C|$, we know that $|C| \leq \frac{1}{2}|I|$. Using $\phi = \frac{2\pi}{13\beta}$ implies $\beta \sin\phi \leq \frac{2\pi}{13} < \frac{1}{2}$. Combining these, we see that $|I| \geq \beta|I|\sin\phi + |C|$, so, I blocks all lines in L_C^i. □

Lemma 9. *Let C_1 be a cell that is marked in T_i by a convex and β-thick region R_1, and let C_2 be a descendant of C_1 that is marked in T_i by a convex and β-thick region R_2. Then there cannot be a descendant C_3 of C_2 that intersects R_1.*

Proof. Let R_2 and R_1 be convex fat regions which mark cells C_2 and C_1 respectively. Then there is a point $p_2 \in R_2 \cap C_2$. Suppose for contradiction that R_1 intersects C_3; that is, there exists a point $p_1 \in R_1 \cap C_3$. Let r and s be two parallel rays from p_1 and p_2 in some direction $\phi \in \Phi_i$. Note that rays r and s are both in $L_{C_2}^i$. Therefore each ray must intersect both R_1 and R_2 by Claim 4.1. Since each region R_1 and R_2 is convex, their intersection with each ray r (or s) is a single line segment, denoted r_1 and r_2 (s_1 and s_2) respectively. Moreover, since R_1 and R_2 are disjoint, the segments r_1 and r_2 (s_1 and s_2) are also disjoint (see Figure 7).

Since $p_1 \in R_1$, r_1 must come before r_2 on the ray r. Similarly, s_2 must come before s_1 on the ray s. Moreover, R_1 is convex, and thus the convex quadrilateral defined by r_1, s_1 is completely contained in R_1, and likewise $r_2, s_2 \subseteq R_2$. These two quadrilaterals must intersect, which is a contradiction because R_1 and R_2 are disjoint. Therefore there is no point $p_1 \in R_1 \cap C_3$. □

Queries. Given a query point q, we want to find out which region (if any) contains q. We begin by performing a point location query for q in the quadtree Q. By Lemma 3 we can find the leaf cell C in the quadtree which contains q in $O(\log n)$ time using the edge-oracle tree.

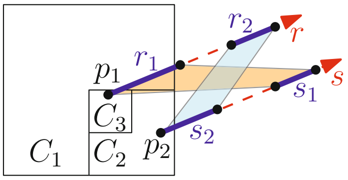

Fig. 7. Illustration of Lemma 9

By Lemma 8, there can only be $O(\beta)$ regions which intersect C. All regions of size at most $2\beta|C|$ will have tagged C with a pointer to themselves, and are immediately available from C. Moreover, we can find all regions of size at least $2\beta|C|$ in $O(\beta \log n)$ time by querying the marked-ancestor structures. We compare each region to our query point, and determine which region (if any) intersects the query point in $O(\beta)$ time. Thus, we can answer the query in total time $O(\beta \log n)$.

Updates. We only store the representative points of the regions in the quadtree. Thus, when performing a local update, it is sufficient to find the new location for the region's representative point, and then update the quadtree, tags, marked-ancestor trees, and edge-oracle trees accordingly.

Given a pointer to a region R, we replace it by another region R' that is ρ-similar to R for any arbitrary parameter $\rho \geq 1$. Let p and p' be the representative points of R and R', respectively. We find the leaf cell of Q containing p' by going up in the quadtreee until the size of the cell we are in is similar to the distance to p', then using level-links to find the ancestor of p' of similar size, and then going back down.

Lemma 10. *The distance in Q between the leaf C containing p and the leaf C' containing p' is at most $O(\log(\rho\beta))$.*

Proof. Recall that by definition, $|R \cup R'| \leq \rho \min\{|R|, |R'|\}$, and by Lemma 6, each region is stored in a quadtree cell proportional to its size, i.e. $|C| \geq \frac{|R|}{4\beta}$. Thus, $|C| \geq \frac{|R \cup R'|}{4\beta\rho}$, and likewise for $|C|'$. Hence, to find C' from C, we move up at most $\log(\beta\rho)$ levels in the quadtree to find a cell of size $\Omega(|R \cup R'|)$, then follow $O(1)$ level-link pointers to find a large cell containing p'. Finally, we move down at most $\log(\beta\rho)$ levels to find C'. $\square$

We must also update the quadtree to reflect the new position of the representative point. By Lemma 2, we can delete p, insert p', and perform the corresponding rebalancing of the quadtree in $O(1)$ worst case time.

A local update replaces an old region R by a new region R' which is ρ-similar to R, but may overlap different quadtree cells than R. Therefore we may require updates to the marked-ancestor structure. Let C be the quadtree cell containing R's representative point. After the update, R' must only intersect $O(\beta)$ quadtree cells which are similar in size to C by Lemma 7. For each of these cells, we test the direction of the representative point of R' and mark it in the corresponding marked-ancestor tree. We also unmark cells which corresponded to the old region R. These updates can be performed in $O(\log\log n)$ time per marked-ancestor structure. We must also remove tags from all cells in S_R and add tags to cells in $S_{R'}$. However, given C and C', this takes $O(\beta)$ time by Lemma 7. By Lemma 3 we can also update the edge-oracle tree in $O(1)$ time.

Theorem 2. *A set of n disjoint convex β-thick objects of constant combinatorial complexity in $\mathbb{R}^2$ can be maintained in a $O(\beta n)$ size data structure that supports insertion, deletion and point location queries in $O(\beta \log n)$ time, and ρ-similar updates in $O(\beta \log\log n + \log(\beta\rho))$ time. All time bounds are worst-case, and the data structure can be implemented on a real-valued pointer machine.*

We can extend the result to non-convex fat regions, by cutting them into convex pieces. This approach only works for polygonal objects, since non-polygonal objects cannot always be partitioned into a finite number of convex pieces. For polygonal objects, we use a theorem by van Kreveld:

Theorem 3 (from [15]). *A δ-wide simple polygon P with n vertices can be partitioned in $O(n \log^2 n)$ time into $O(n)$ β-wide quadrilaterals and triangles, where $\beta = \min\{\delta, 1 - \frac{1}{2}\sqrt{3}\}$.*

We conclude:

Theorem 4. *A set of n disjoint polygonal δ-wide objects of constant combinatorial complexity in $\mathbb{R}^2$ can be maintained in a $O(\delta n)$ size data structure that supports insertion, deletion and point location queries in $O(\delta \log n)$ time, and ρ-similar updates in $O(\delta \log\log n + \log(\delta\rho))$ time. All time bounds are worst-case, and the data structure can be implemented on a real-valued pointer machine.*

5 Discussion

We have shown that given a set of regions in $\mathbb{R}^1$ or $\mathbb{R}^2$ fitting some modest assumptions, we can perform local updates in $\mathbb{R}^1$ in $O(1)$ time and in $\mathbb{R}^2$ in $O(\log\log n)$ time respectively. The following are open problems for future research. Can we also handle local updates in $\mathbb{R}^2$ in $O(1)$ time? Can we relax our assumption that the regions must not intersect each other? Can we adapt our techniques to handle regions in $\mathbb{R}^3$ or higher dimensions?

Acknowledgments. Work on this paper has been partially supported by the Office of Naval Research under MURI grant N00014-08-1-1015. M.L. is further supported by the Netherlands Organisation for Scientific Research (NWO) under grant 639.021.123.

References

1. Alstrup, S., Husfeldt, T., Rauhe, T.: Marked ancestor problems. In: Proc. 39th Symp. on Foundations of Computer Science, pp. 534–543 (1998)
2. Arge, L., Brodal, G.S., Georgiadis, L.: Improved dynamic planar point location. In: Proc. 47th Symp. on Foundations of Computer Science, pp. 305–314 (2006)
3. Bentley, J.L.: Solutions to Klee's rectangle problems. Technical report, Carnegie-Mellon Univ., Pittsburgh, PA (1977)
4. Bern, M., Eppstein, D., Gilbert, J.: Provably good mesh generation. J. Comput. Syst. Sci. 48(3), 384–409 (1994)
5. Cheng, S.W., Janardan, R.: New results on dynamic planar point location. SIAM J. Comput. 21(5), 972–999 (1992)
6. Berg, M.d., Gray, C.: Vertical ray shooting and computing depth orders for fat objects. In: SODA, pp. 494–503. ACM Press (2006)
7. Dobkin, D.P., Lipton, R.J.: Multidimensional searching problems. SIAM J. Comput. 5(2), 181–186 (1976)
8. Finkel, R.A., Bentley, J.L.: Quad trees: A data structure for retrieval on composite keys. Acta Inform. 4, 1–9 (1974)
9. Fleischer, R.: A simple balanced search tree with o(1) worst-case update time. In: Ng, K.W., Balasubramanian, N.V., Raghavan, P., Chin, F.Y.L. (eds.) ISAAC 1993. LNCS, vol. 762, pp. 138–146. Springer, Heidelberg (1993)

10. Giora, Y., Kaplan, H.: Optimal dynamic vertical ray shooting in rectilinear planar subdivisions. ACM Trans. Algorithms 5(3), 28:1–28:51 (2009)
11. Husfeldt, T., Rauhe, T.: Lower bounds for dynamic transitive closure, planar point location, and parentheses matching. Nordic J. Computing 3 (1996)
12. Löffler, M., Simons, J.A., Strash, D.: Dynamic planar point location with sub-logarithmic local updates. Arxiv report, arXiv:1204.4714 [cs.CG] (April 2012)
13. Nekrich, Y.: Data structures with local update operations. In: Gudmundsson, J. (ed.) SWAT 2008. LNCS, vol. 5124, pp. 138–147. Springer, Heidelberg (2008)
14. Sarnak, N., Tarjan, R.E.: Planar point location using persistent search trees. Commun. ACM 29, 669–679 (1986), `http://doi.acm.org/10.1145/6138.6151`, doi:10.1145/6138.6151
15. van Kreveld, M.: On fat partitioning, fat covering, and the union size of polygons. Comput. Geom. Theory Appl. 9(4), 197–210 (1998)