An Output-Sensitive Algorithm for Computing the Union of Cubes and Fat Boxes in 3D

Pankaj K. Agarwal
Department of Computer Science, Duke University, Durham, NC, USA
pankaj@cs.duke.edu

Alex Steiger
Department of Computer Science, Duke University, Durham, NC, USA
asteiger@cs.duke.edu

Abstract

Let $\mathcal{C}$ be a set of n axis-aligned cubes of arbitrary sizes in $\mathbb{R}^3$. Let $\mathcal{U}$ be their union, and let κ be the number of vertices on $\partial\mathcal{U}$; κ can vary between $O(1)$ and $O(n^2)$. We show that $\mathcal{U}$ can be computed in $O(n\log^3 n+\kappa)$ time if $\mathcal{C}$ is in general position. The algorithm also computes the union of a set of fat boxes (i.e., boxes with bounded aspect ratio) within the same time bound. If the cubes in $\mathcal{C}$ are congruent or have bounded depth, the running time improves to $O(n\log^2 n)$, and if both conditions hold, the running time improves to $O(n\log n)$.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases union of cubes, fat boxes, plane-sweep

Digital Object Identifier 10.4230/LIPIcs.ICALP.2021.10

Category Track A: Algorithms, Complexity and Games

Funding Partially supported by NSF grants IIS-18-14493 and CCF-20-07556.

1 Introduction

Let $\mathcal{C}$ be a set of n axis-aligned cubes of arbitrary sizes in $\mathbb{R}^3$. Let $\mathcal{U}:=\mathcal{U}(\mathcal{C})$ be their union, and let κ be the number of vertices on $\partial\mathcal{U}$. It is known that $\kappa=\Theta(n^2)$ in the worst case, though it is linear or near-linear in many special cases. For example, $\kappa=O(n)$ if the cubes in $\mathcal{C}$ have roughly the same size [6] or if they have bounded depth (i.e., any point in $\mathbb{R}^3$ lies in $O(1)$ cubes) [14]. If their sizes are drawn independently from an arbitrary distribution, the expected complexity of their union is $O(n\log^2 n)$ [3]. A natural problem is to develop an algorithm for computing $\partial\mathcal{U}$, by which we mean compute its vertices, edges, and faces. Although $\mathcal{U}$ can be computed in $O(n^2\log n)$ time by computing $\partial\mathcal{U}$ on each face of every cube of $\mathcal{C}$, an output-sensitive algorithm with $O(n\log n+\kappa)$ running time has remained elusive. In this paper we present an algorithm that almost matches this running time.

Related work. Motivated by VLSI design and other applications, the problem of computing the union of a set of axis-aligned rectangles in $\mathbb{R}^2$ and its variants have been studied since the 1970's; see [17]. An optimal $O(n\log n+\kappa)$-time algorithm was presented by Güting [12].

A closely related problem, which also has been studied extensively, is the so-called *Klee's measure problem*, which asks to compute the *volume* of the union of axis-aligned boxes in $\mathbb{R}^d$. For $d=2$, Bentley presented an $O(n\log n)$-time algorithm for this problem, which extends to higher dimensions and computes the volume in $O(n^{d-1}\log n)$ time. For $d\geq 3$, Overmars and Yap [16] gave an $O(n^{d/2}\log n)$-time algorithm. The running time was improved to $O(n^{d/2})$ by Chan [8]. It was an open question whether a faster algorithm exists if the input boxes are hypercubes or fat. Agarwal *et al.* [2] described an $O(n^{4/3}\log n)$-time algorithm for cubes in 3D, which was subsequently improved to $O(n\log^4 n)$ in [1]. Bringmann [7]

presented an $O(n^{(d+2)/3})$-time algorithm for fat boxes in $\mathbb{R}^d$, which was later improved to $O(n^{(d+1)/3}\,\mathrm{polylog}(n))$ in [8]; see also [19].

Despite extensive work on Klee's measure problem, relatively less is known about computing the union boundary. For example, no $O(n \log n)$-time algorithm is known even for computing the union of unit cubes in $\mathbb{R}^3$. For $d = 3$, the Overmars-Yap algorithm can be adapted to compute the boundary of the union of boxes in $O((n^{3/2} + \kappa) \log n)$ time, where κ is the output size. But it is not obvious whether the algorithm in [1] can be adapted to compute the union of axis-aligned cubes in $O((n + \kappa)\,\mathrm{polylog}(n))$ time. Agarwal *et al.* [4] have presented a randomized algorithm that computes the union of congruent cubes in $\mathbb{R}^3$ in $O(n^{1+\varepsilon})$ expected time, for any constant $\varepsilon > 0$.

A related line of work is to partition the union (or its complement) into few axis-aligned boxes. Chew *et al.* [9] describe an algorithm to compute a partition of the union of n congruent cubes into $O(n)$ axis-aligned boxes in $O(n \log n)$ time, assuming that the union has already been computed[1]. For a special case of orthants in $\mathbb{R}^d$, Kaplan *et al.* [15] describe an algorithm to partition the union of n such orthants into $O(\kappa)$ axis-aligned (semiunbounded) boxes in $O((n + \kappa) \log^{d-1} n)$ time, for any $d \geq 1$, where κ is the union complexity.

Our results. The main result of the paper is an output-sensitive algorithm to compute $\mathcal{U}$. That is, our algorithm computes the vertices, edges, and 2D faces of $\partial\mathcal{U}$. For each 2D face f, it computes the components of ∂f. We say that $\mathcal{C}$ is in *general position* if any plane contains the boundary face of at most one cube in $\mathcal{C}$. Our algorithm assumes $\mathcal{C}$ to be in general position. Although it can be extended to degenerate configurations using symbolic perturbation techniques (e.g., [10]), the running time depends on the union complexity of the perturbed configuration.

▶ **Theorem 1.** *Given a set $\mathcal{C}$ of n axis-aligned cubes in $\mathbb{R}^3$, $\mathcal{U}(\mathcal{C})$ can be computed in $O(n \log^3 n + \kappa)$ time if $\mathcal{C}$ is in general position. If $\mathcal{C}$ is not in general position, the running time is $O(n \log^3 n + \widehat{k})$, where $\widehat{k}$ is the maximum complexity of $\mathcal{U}(\mathcal{C})$ under an infinitesmal perturbation.*

A 3D box is *fat* if its aspect ratio (i.e., the ratio of its longest side length and its shortest side length) is bounded by a constant. A fat box can be decomposed into $O(1)$ (possibly intersecting) cubes. Replacing fat boxes in general position with $O(1)$ such cubes then perturbing them into general position increases the union complexity at most by a constant factor. From Theorem 1, we have the following:

▶ **Corollary 2.** *Given a set $\mathcal{B}$ of n axis-aligned fat boxes in $\mathbb{R}^3$, where the aspect ratio of each box is bounded by a constant, $\mathcal{U}(\mathcal{B})$ can be computed in $O(n \log^3 n + \kappa)$ time, assuming $\mathcal{B}$ is in general position. If $\mathcal{B}$ is not in general position, the running time is $O(n \log^3 n + \widehat{k})$, where $\widehat{k}$ is the maximum complexity of $\mathcal{U}(\mathcal{B})$ under an infinitesmal perturbation.*

For some special cases, simpler algorithms compute the union slightly more efficiently: when all cubes in $\mathcal{C}$ are congruent or when they have bounded *depth* (i.e., any point in $\mathbb{R}^3$ is contained in at most c cubes of $\mathcal{C}$, for a constant $c > 0$). In both cases, $\kappa = O(n)$. Since the output size is always linear after perturbing the cubes to be in general position, we do not need the general-position assumption here.

[1] Theorem 1 in [9] suggests that the union $\mathcal{U}(\mathcal{C})$ of n unit axis-aligned cubes $\mathcal{C}$ in $\mathbb{R}^3$ can be computed in $O(n \log n)$, but no such algorithm is presented in the paper. It shows that given $\mathcal{U}(\mathcal{C})$, it can be decomposed into $O(n)$ boxes in $O(n \log n)$ time [11].

▶ **Theorem 3.** *Let $\mathcal{C}$ be a set of n axis-aligned cubes in $\mathbb{R}^3$. If all cubes in $\mathcal{C}$ are congruent or they have bounded depth, then $\mathcal{U}(\mathcal{C})$ can be computed in $O(n \log^2 n)$ time. If the cubes in $\mathcal{C}$ are congruent and have bounded depth, then $\mathcal{U}(\mathcal{C})$ can be computed in $O(n \log n)$ time.*

Analogous to Corollary 2, we obtain the following:

▶ **Corollary 4.** *Given a set $\mathcal{B}$ of n axis-aligned fat boxes in $\mathbb{R}^3$. If the ratio of the largest to the smallest size box is bounded by a constant or they have bounded depth, then $\mathcal{U}(\mathcal{B})$ can be computed in $O(n \log^2 n)$ time. If both conditions hold, then the running time is $O(n \log n)$.*

At a high level, the general approach of our algorithm is similar to Agarwal's [1] to compute the volume of the union of 3D cubes, but ours is considerably simpler and more efficient. We reduce the problem to maintaining the union of a set of squares in the plane under insertions and deletions, which are the xy-projections of the faces of cubes in $\mathcal{C}$. At the core of the data structure in [1] is a hierarchical decomposition of the plane using a variant of a kd-tree. In constrast, our data structure is a variant of a quadtree whose regions are squares. Using this property — having square regions — we crucially circumvent much of the intracacies in [1] and attain a simpler algorithm. We use a sweep-line algorithm to compute the changes in the union-boundary of squares and rely on auxiliary data structures, which are also somewhat simpler than in [1] to perform this sweep efficiently. Finally, we are also able to simplify and improve the algorithm because we can charge time to the $O(\kappa)$ vertices reported.

Roadmap of the paper. As a warm-up, we first present in Section 2 an algorithm for a special case where the spread of the xy-projections of the vertices of the cubes in $\mathcal{C}$ (regarded as a 2D point set) is (polynomially) bounded, i.e., the ratio of the distance between the farthest and closest pairs of such points in $\mathbb{R}^2$ is bounded by n^c, for some constant $c > 0$. In Section 3 we describe how to remove this assumption to obtain our main result (Theorem 1). Finally, we describe the more efficient algorithms for congruent cubes and cubes with bounded depth (Theorem 3) in Section 4.

2 Algorithm for the Bounded Spread Case

Let $\mathcal{C} := \{C_1, \ldots, C_n\}$ be a set of n axis-aligned cubes in $\mathbb{R}^3$ in general position. We assume that the spread of the xy-projections of the vertices of $\mathcal{C}$ is polynomially bounded. For any set A of objects (e.g. segments in $\mathbb{R}$, squares in $\mathbb{R}^2$, or cubes in $\mathbb{R}^3$), let $\mathcal{U}(A)$ denote the union of the objects in A, and let $V(A)$ be the vertices of $\mathcal{U}(A)$. Set $\mathcal{U} := \mathcal{U}(\mathcal{C})$. For any 3D object a, let $a^{\downarrow}$ be the xy-projection of a.

In this section, we describe an algorithm to compute the boundary of $\mathcal{U}$, denoted by $\partial\mathcal{U}$, namely its vertices, edges, and faces. Once the vertices of $\partial\mathcal{U}$ have been computed, the edges and faces can be computed using standard techniques, so we first focus only on computing the vertices and remark at the end of the section how to extend it to compute edges and faces of $\partial\mathcal{U}$.

2.1 Overview of the algorithm

We first introduce some notation. For a cube $C_i \in \mathcal{C}$, let $S_i := C_i^{\downarrow}$ be the xy-projection of C_i, which is an axis-aligned square in $\mathbb{R}^2$. Let $z_1 < \ldots < z_{2n}$ be the z-coordinates of the vertices of cubes in $\mathcal{C}$, sorted in decreasing order. For all i with $1 \le i \le 2n$, let $\mathcal{C}_i \subseteq \mathcal{C}$ be the set of

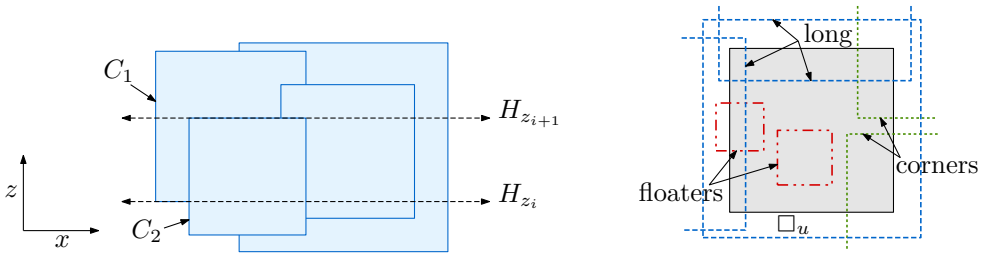

Figure 1 (left) A 2D view of $\mathcal{C}$. The bottom (resp. top) face of cube C_1 (resp. C_2) lies on the xy-plane $H_{z_i} : z = z_i$ (resp. $H_{z_{i+1}} : z = z_{i+1}$). (right) Various long and short squares at node u.

cubes whose z-spans cover the interval (z_i, z_{i+1}), and let $\mathcal{S}_i = \{S_j \mid C_j \in \mathcal{C}_i\}$. Let V_i denote the set of vertices of $\mathcal{U}(\mathcal{S}_i)$.

Note that every vertex of $\mathcal{U}$ is the intersection of three orthogonal faces of cubes in $\mathcal{C}$ — in particular, every vertex lies on a xy-face of some cube, i.e., the z-coordinate of every vertex is one of the z_i's, and is incident to a z-edge of $\mathcal{U}$. Therefore, our high-level approach is to sweep a horizontal plane H in the $(+z)$-direction, from z_1 to z_{2n}, and maintain $H \cap \mathcal{U}$ in the process. We stop the sweep at each z_i and report the vertices of $\mathcal{U}$ that are incident on the face of the cube of $\mathcal{C}$ lying in the plane H_{z_i}.

For any value $a \in (z_i, z_{i+1})$, let $\mathcal{U}_a := \mathcal{U} \cap H_a$. Then we have

$$\mathcal{U}_a = \mathcal{U}\big(\{H_a \cap C \mid C \in \mathcal{C}_i\}\big) = \mathcal{U}(\{S_i \times \{a\} \mid S_i \in \mathcal{S}_i\}) = \mathcal{U}(\mathcal{S}_i) \times \{a\}$$

so the combinatorial structure of $\mathcal{U}$, and $\mathcal{U}(\mathcal{S}_i) = \mathcal{U}_a^{\downarrow}$, does not change in $[z_i, z_{i+1})$. See Figure 1. Furthermore, each vertex (x_0, y_0, a) of $\mathcal{U}_a$ is the intersection point of a z-edge e of $\mathcal{U}$ with H_a and $(x_0, y_0) \in V_i$. Let p^- (resp. p^+) be the lower (resp. upper) endpoint of e. Let $z_i := z(p^-)$ and $z_j := z(p^+)$, where $i < j$. Then $(p^-)^{\downarrow} \in V_i \setminus V_{i-1}$, and $(p^+)^{\downarrow} \in V_{j-1} \setminus V_j$. Conversely, for any i with $1 \le i \le 2n$, every vertex of $\Delta V_i := V_i \oplus V_{i-1}$ is the projection of an endpoint of a z-edge of $\mathcal{U}$ and thus a vertex of $\mathcal{U}$, where $\oplus$ is symmetric difference. (So that ΔV_1 is well-defined, set $\Delta V_0 := \varnothing$.) Thus, to compute the vertices of $\mathcal{U}$, we report ΔV_i when the plane H stops at z_i, for all i, $1 \le i \le 2n$.

We report the vertices of ΔV_i using a data structure $\mathbb{T}$ that stores $\mathcal{S}_i$ and implicitly maintains $\mathcal{U}(\mathcal{S}_i)$ as we sweep H. A square S_j is inserted into $\mathcal{S}_i$ when H reaches the bottom face of C_j, and it is deleted from $\mathcal{S}_i$ when H reaches the top face of C_j. Let $|A|$ denote the length of the longest side length of A, where A is a box or rectangle. Since C_j's are cubes, the sequence of updates satisfies the following property:

(P1): Let $C_u, C_v \in \mathcal{C}$ be two cubes such that $|C_u| < |C_v|$ and S_u is inserted before S_v. Then S_u is also deleted before S_v.

When the sweep stops at z_i, the insertion or deletion of a square to $\mathcal{S}_{i-1}$ (to obtain $\mathcal{S}_i$) into $\mathbb{T}$ is performed in $O(\log^3 n)$ amortized time, and ΔV_i is reported in $O(\log^3 n + |\Delta V_i|)$ amortized time. Thus, the sweep takes $O(\sum_{i=1}^{2n} \log^3 n + \sum \Delta V_i) = O(n \log^3 n + \kappa)$ time. With $O(n \log n)$ additional preprocessing to initialize $\mathbb{T}$ before the sweep, and $O(n \log n + \kappa)$ postprocessing to compute the edges and faces of $\mathcal{U}$ using the vertices reported during the sweep, the algorithm takes $O(n \log^3 n + \kappa)$ overall time, proving Theorem 1 for the bounded-spread case.

2.2 Reporting ΔV_i

Next, we describe our dynamic data structure $\mathbb{T}$ that stores a set $\mathcal{S}$ of squares in $\mathbb{R}^2$ and implicitly maintains $\mathcal{U}(\mathcal{S})$. After each update — an insertion or deletion of a square — it reports $\Delta V := V(\mathcal{S}^{\text{new}}) \oplus V(\mathcal{S}^{\text{old}})$, where $\mathcal{S}^{\text{old}}$ (resp. $\mathcal{S}^{\text{new}}$) is $\mathcal{S}$ before (resp. after) the update operation. Let $\mathsf{X} \subset \mathbb{R}^2$ be the set of points corresponding to the xy-projections of vertices of cubes in $\mathcal{C}$. Recall that the spread of X is polynomially bounded. Let $\square_0$ be a square containing X. $\mathbb{T}$ is a quadtree built on X with $\square_0$ being the square associated with the root node of $\mathbb{T}$. Without loss of generality, we assume that no point of X lies on the boundary of a square $\square_u$ for any node $u \in \mathbb{T}$.

Each node $u \in \mathbb{T}$ is associated with a square $\square_u$. For the root node r, $\square_r = \square_0 \supset \mathsf{X}$. If $|\mathsf{X} \cap \square_u| \leq 1$, u is a leaf, otherwise $\square_u$ is partitioned into four congruent squares, each associated with a child of u. Set $\mathsf{X}_u := \mathsf{X} \cap \square_u$. The height of $\mathbb{T}$ is $O(\log n)$. A square S that intersects the region $\square_u$, for a node $u \in \mathbb{T}$, is *long* at u if no vertex of S lies in $\square_u$, and S is *short* at u otherwise. A short square S at node u is called a *floater square* if at least two vertices of S lie in $\operatorname{int}(\square_u)$, and a *corner square* otherwise. A corner square S contains exactly one vertex of $\square_u$. See Figure 1 (right). For each node $u \in \mathbb{T}$, let $\mathcal{L}_u \subseteq \mathcal{S}$ (resp. $\mathcal{S}_u \subseteq \mathcal{S}$) be the set of long (resp. short) squares at u, and let $\mathcal{L}^*_u := \mathcal{L}_u \setminus \mathcal{L}_{p(u)} = \mathcal{L}_u \cap \mathcal{S}_{p(u)}$, where $p(u)$ is the parent node of u in $\mathbb{T}$; $\mathcal{L}_{\text{root}} := \varnothing$. Let $\mathcal{F}_u \subseteq \mathcal{S}_u$ (resp. $\mathcal{R}_u \subseteq \mathcal{S}_u$) denote the set of floater (resp. corner) squares at u. Note that any square $S \in \mathcal{S}$ is short at no more than four nodes in any level of $\mathbb{T}$. Hence S is short at $O(\log n)$ nodes of $\mathbb{T}$.

At each node $u \in \mathbb{T}$, we maintain $\mathcal{L}^*_u$ and $\mathcal{S}_u$. A long square $S \in \mathcal{L}^*_u$ contains at least one edge of $\square_u$; it may contain all four edges of $\square_u$ if $\square_u \subseteq S$. We partition $\mathcal{L}^*_u$ into four sets: for each edge $e \in \square_u$, let $\mathcal{L}^*_{u,e} \subseteq \mathcal{L}^*_u$ be the set of squares that contain e. If $\square_u \subseteq S$, then S is assigned only to the set associated with the top edge of $\square_u$. Suppose e is the top edge of $\square_u$. Then the bottom edges of squares in $\mathcal{L}^*_{u,e}$ intersect $\square_u$ or lie below $\square_u$. We store $\mathcal{L}^*_{u,e}$ in a red-black tree, sorted in increasing order of the distances of their bottom edges from e (i.e., in decreasing order of their y-coordinates). We similarly store $\mathcal{L}^*_{u,e}$ for the other three edges e of $\square_u$. Let $\mathcal{E}^x_u$ (resp. $\mathcal{E}^y_u$) be the sequences of x-edges (resp. y-edges) of $\mathcal{F}_u$ in increasing order of their y-coordinates (resp. x-coordinates). We maintain $\mathcal{E}^x_u, \mathcal{E}^y_u$ using red-black trees.

For each edge e of $\square_u$, we store a value ℓ_e that maintains the position of a sweep-line L_e associated with the edge e. The roles of ℓ_e and L_e will become clear in the insertion and deletion procedures. The value of ℓ_e (and the sweep-line) changes dynamically. Initially, for each x-edge (resp. y-edge) e of $\square_u$, ℓ_e is the y-coordinate (resp. x-coordinate) of e, and it always lies in the y-span (resp. x-span) of $\square_u$. If e is an x-edge (resp. y-edge), then let $\mathcal{I}_e$ be the set of x-projections (resp. y-projections) of the floater squares in $\mathcal{F}_u$ whose y-spans (resp. x-spans) contain ℓ_e. $\mathcal{I}_e$ changes dynamically with ℓ_e.

We also maintain two secondary structures for each edge e of $\square_u$:

- *Wall data structure* $\mathbb{W}_e(\mathcal{I}_e)$: It supports the following two operations:
  - Insert/Delete(I): Insert or delete an interval I to $\mathcal{I}_e$.
  - Report-Holes(ζ): For a query interval ζ, return the endpoints of $\zeta \setminus \operatorname{int}(\mathcal{U}(\mathcal{I}_e))$. If an endpoint x of ζ does not lie in $\operatorname{int}(\mathcal{U}(\mathcal{I}_e))$, x is also returned.

  The Insert, Delete, and Membership operations take $O(\log n)$ time and Report-Holes takes $O(\log n + \kappa)$ time, where κ is the number of points reported.
- *Corner data structure* $\mathbb{C}_e(\mathcal{R}_u, \mathcal{I}_e)$: It supports the following four operations:
  - Insert/Delete(R): Insert or delete a corner square R to $\mathcal{R}_u$.
  - Insert/Delete(I): Insert or delete an interval I to $\mathcal{I}_e$.
  - Report-Hole(σ): Given a query axis-aligned segment $\sigma \subset \square_u$, return the (at most one) interval of $\sigma \setminus \operatorname{int}(\mathcal{U}(\mathcal{R}_u))$.

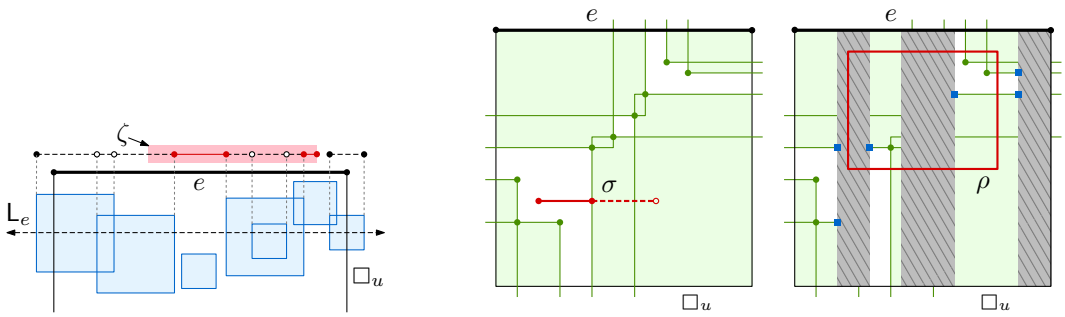

Figure 2 (left) An illustration of call Report-Hole(ζ) to $\mathbb{W}_e$. The x-projections of floater squares intersecting L_e that compose $\mathcal{I}_e$. The intervals of $\mathcal{U}(\mathcal{I}_e)$ are dashed. The solid red intervals in ζ are the answer to the query. (right) Illustrations of calls Report-Hole(σ) and Report-Vertices(ρ) to $\mathbb{C}_e$. The vertices of $\mathcal{U}(\mathcal{R}_u)$ are shown as green circles, and the vertices of $\mathcal{U}(\mathcal{R}_u \cup \mathcal{W}_e)$ that are the intersection points of a "strip" in $\mathcal{W}_e$ (gray hatched) and an edge of $\mathcal{U}(\mathcal{R}_u)$ are shown as blue squares. The solid portion of σ is the answer to Report-Hole(σ), and the two blue vertices and three green vertices (inside ρ) are the answer to Report-Vertices(ρ).

- Report-Vertices(ρ): Given a query rectangle $\rho \subseteq \square_u$, return $V(\mathcal{R}_u \cup \mathcal{W}_e) \cap \rho$, where $\mathcal{W}_e := \{I \times \mathbb{R} \mid I \in \mathcal{I}_e\}$ if e is an x-edge and $\mathcal{W}_e := \{\mathbb{R} \times I \mid I \in \mathcal{I}_e\}$ if e is a y-edge. Namely, the vertices of $\mathcal{U}(\mathcal{R}_u)$ that do not lie in $\mathcal{U}(\mathcal{W}_e)$ or the intersection points of the edges of $\mathcal{U}(\mathcal{R}_u)$ and $\mathcal{U}(\mathcal{W}_e)$ that lie in ρ are reported.

An Insert/Delete takes $O(\log^2 n)$ time, Report-Hole takes $O(\log n)$ time, and Report-Vertices takes $O(\log n + \kappa)$ time, where κ is the number of vertices reported.

See Figure 2. We describe these data structures in Section 2.4. For now, we assume that, for all four edges of $\square_u$ and for all nodes $u \in \mathbb{T}$, $\mathbb{W}_e$, and $\mathbb{C}_e$ are at our disposal.

For each square $S \in \mathcal{S}$, let $\Lambda(S) := \{u \in \mathbb{T} \mid S \in \mathcal{L}^*_u\}$ and $\Sigma(S) := \{u \in \mathbb{T} \mid S \in \mathcal{S}_u\}$. Any square $S \in \mathcal{S}$ is short at no more than four nodes of any fixed level of $\mathbb{T}$. The nodes in $\Sigma(S)$ lie along at most four root-to-leaf paths to the leaves whose squares contain the vertices of S, and the nodes in $\Lambda(S)$ are those children u of nodes in $\Sigma(S)$ for which $S \cap \square_u \neq \varnothing$ and S is not short at u. Since the height of $\mathbb{T}$ is $O(\log n)$, $|\Lambda(S)|, |\Sigma(S)| = O(\log n)$. Finally, let $\Xi(S)$ be $\Lambda(S)$ and the four leaves of $\Sigma(S)$, and set $\Pi(S) := \{S \cap \square_u \mid \square_u \in \Xi(S)\}$. See Figure 3 (left). The following lemma is straightforward.

▶ **Lemma 5.** *For any square S, $\Pi(S)$ is a partition of S into $O(\log n)$ rectangles.*

When we insert or delete a square S, every vertex of ΔV lies in S. Thus, to report ΔV, we report $\Delta V_u := \Delta V \cap \square_u$ for each node $u \in \Xi(S)$. We now describe how to update $\mathbb{T}$ and compute ΔV_u at each node $u \in \Xi(S)$.

Insertion of S. There are four main steps:

(1) At each node $u \in \Lambda(S) \cup \Sigma(S)$, we compute $\rho_u := \mathrm{cl}(\square_u \setminus \mathcal{U}(\mathcal{L}_u))$, as described below.

(2) At each node $u \in \Xi(S)$, we report ΔV_u, which is also described below.

(3) For each node $u \in \Lambda(S)$, we insert S to $\mathcal{L}^*_u$. In particular, if S contains the edge e of $\square_u$, S is inserted into $\mathcal{L}^*_{u,e}$; recall that if $\square_u \subseteq S$, then S is associated with the top edge of $\square_u$.

(4) For each $u \in \Sigma(S)$, we update $\mathcal{S}_u$ and its secondary structures, as follows. If S is a corner square at u, we insert S into all four corner data structures $\mathbb{C}_e$ stored at u for each edge of $\square_u$. If S is a floater, we first insert the x-edges (resp. y-edges) of S which intersect

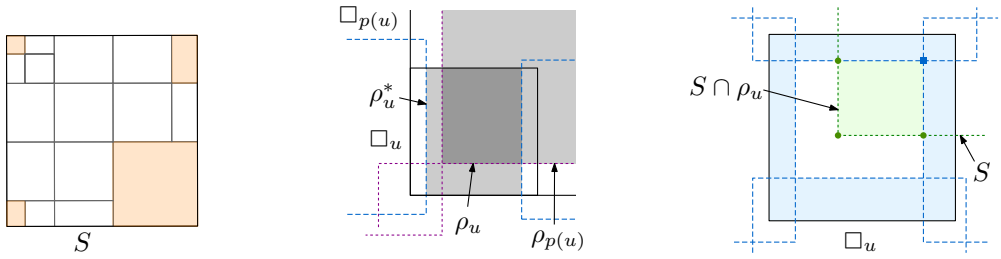

Figure 3 (left) The rectangles of $\Pi(S)$. The shaded (resp. empty) rectangles lie in squares $\square_u$ of nodes u in $\Sigma(S)$ (resp. $\Lambda(S)$). (middle) Computing ρ_u using ρ_u^* and $\rho_{p(u)}$. The squares at the ends of the sequences $\mathcal{L}_{u,e}^*$ for each edge e of $\square_u$ are dashed and the squares of $\mathcal{L}_{p(u)}$ that contribute to $\mathcal{U}(\mathcal{L}_{p(u)})$ are dotted. (right) An example of a square S being inserted at $u \in \Sigma(S)$. The four vertices of ΔV_u are shown, where the blue vertex is old and the rest are new.

$\square_u$ into $\mathcal{E}_u^*$ (resp. $\mathcal{E}_u^y$). Then, for each x-edge (resp. y-edge) e of $\square_u$ such that ℓ_e lies in the y-span (resp. x-span) of S, we insert the x-projection (resp. y-projection) of S to $\mathbb{W}_e$ and $\mathbb{C}_e$.

We now describe the first two steps in more detail. We compute ρ_u at each node $u \in \Lambda(S) \cup \Sigma(S)$, as follows. Observe that $\rho_u \subseteq \square_u$ is a (possibly empty) rectangle. Similarly, $\mathrm{cl}(\square_u \setminus \mathcal{U}(\mathcal{L}_u^*))$ is a rectangle $\rho_u^* \subseteq \square_u$. By definition, $\rho_u = \mathrm{cl}(\square_u \setminus (\mathcal{U}(\mathcal{L}_{p(u)}) \cup \mathcal{U}(\mathcal{L}_u^*))) = \rho_{p(u)} \cap \rho_u^*$. See Figure 3 (middle). Furthermore, for each edge e_i, $1 \le i \le 4$, let S_i be the last square in the sequence $\mathcal{L}_{u,e_i}^*$, and let h_i be the halfplane bounded by the line supporting the edge of S_i that intersects $\square_u$ and does not contain S_i (if $S_i \supseteq \square_u$, then choose the halfplane bounded by the edge of $\square_u$ opposite e_i and not containing $\square_u$). Then $\rho_u^* = \bigcap_{i=1}^4 h_i \cap \square_u$. Hence, ρ_u^* can be computed in $O(1)$ time. Also, with $\rho_{p(u)}$ at our disposal $\rho_u = \rho_{p(u)} \cap \rho_u^*$ can be computed in $O(1)$ time. Thus, using a top-down traversal of $\mathbb{T}$, starting at the root node r (with $\mathcal{L}_r = \varnothing$) and ending at the nodes of $\Xi(S)$, we compute ρ_u at each visited node; $O(\log n)$ nodes are visited in this process, so the step takes $O(\log n)$ time overall. This completes step (1).

Next, let u be a node of $\Xi(S)$. We report ΔV_u as follows. If $u \in \Sigma(S)$, then $\mathcal{S}_u = \varnothing$ (before the insertion of S) and hence ΔV_u can be reported in $O(1)$ time; see Figure 3 (right). We now focus on reporting ΔV_u when $u \in \Lambda(S)$. For sake of concreteness, suppose S contains the top edge of $\square_u$. Let S_u^{exp} be the rectangle $S \cap \rho_u = \mathrm{cl}((S \cap \square_u) \setminus \mathcal{U}(\mathcal{L}_u)))$, i.e., the portion of $S \cap \square_u$ that is left "exposed" by the squares of $\mathcal{L}_u$. All vertices of ΔV_u lie in S_u^{exp}. Note that S_u^{exp} may be empty, e.g., if $\square_u \subseteq \mathcal{U}(\mathcal{L}_u)$ or $S \cap \rho_u = \varnothing$, in which case $\Delta V_u = \varnothing$ and there is nothing to do; see Figure 4 (middle). If $S_u^{\mathsf{exp}} \neq \varnothing$, we sweep a line L_e in the $(-y)$-direction from the top edge of S_u^{exp} (defined by $\mathcal{U}(\mathcal{L}_u)$ or $\square_u$) to its bottom edge (defined by S, $\square_u$, or $\mathcal{U}(\mathcal{L}_u)$), and report all vertices of ΔV_u in the process by using the secondary data structures as described in Section 2.3. Recall that ℓ_e keeps track of the position of L_e.

Deletion of S. Intuitively, the deletion is "undoing" the insertion of S.

(1) At each node $u \in \Lambda(S) \cup \Sigma(S)$, we compute $\rho_u := \mathrm{cl}(\square_u \setminus \mathcal{U}(\mathcal{L}_u \setminus \{S\}))$ as in the insertion case.

(2) At each node $u \in \Xi(S)$, we report ΔV_u, as described below.

(3) For each node $u \in \Lambda(S)$, we delete S from $\mathcal{L}_u^*$.

(4) For each node $u \in \Sigma(S)$, we update $\mathcal{S}_u$ and its secondary structures, as follows. If S is a corner square at u, we delete S from all four corner data structures $\mathbb{C}_e$ stored at u for each edge e of $\square_u$. If S is a floater, we first delete the x-edges (resp. y-edges) of S which

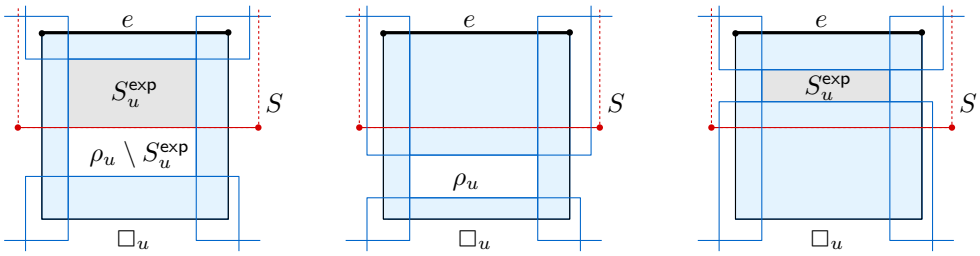

Figure 4 Various cases for S_u^{exp} when $\rho_u \neq \varnothing$: (left) $S_u^{\mathsf{exp}} \subset \rho_u$. (middle) $S_u^{\mathsf{exp}} = \varnothing$. (right) $S_u^{\mathsf{exp}} = \rho_u$.

intersect $\square_u$ from $\mathcal{E}_u^*$ (resp. $\mathcal{E}_u^y$). Then, for each x-edge (resp. y-edge) e of $\square_u$, we delete the x-projection (resp. y-projection) of S from $\mathbb{W}_e$ and $\mathbb{C}_e$ if ℓ_e lies in the y-span (resp. x-span) of S.

Let S_u^{exp} be the rectangle $S \cap \rho_u = \mathrm{cl}(S \cap \square_u) \setminus \mathcal{U}(\mathcal{L}_u \setminus \{S\}))$ i.e., the portion of $S \cap \square_u$ that is left "exposed" by the other squares of $\mathcal{L}_u$. Again, all vertices of ΔV_u lie in S_u^{exp}. We note that S_u^{exp} may be empty because ρ_u is empty or the bottom edge of S lies above the top edge of S_u^{exp}. If $S_u^{\mathsf{exp}} = \varnothing$, $\Delta V_u = \varnothing$ and there is nothing to do. So assume $S_u^{\mathsf{exp}} \neq \varnothing$. We report ΔV_u by sweeping a line L_e in the $(+y)$-direction from the bottom edge of S_u^{exp} (defined by S, $\square_u$, or $\mathcal{U}(\mathcal{L}_u)$), to its top edge (which is the top edge of ρ_u).

Runtime analysis. We now analyze the amortized running time of inserting or deleting a square. Let c be a sufficiently large constant. For each node $u \in \mathbb{T}$, we assign $4c \log^2 n$ credits to each of the four edges of every floater in $\mathcal{F}_u$. These credits will be used to pay part of the cost of reporting ΔV_u (see Lemma 8). Note that S is a floater at $O(\log n)$ nodes. Therefore $O(\log^3 n)$ credits are assigned to a square S. This cost is charged to the insertion of S.

Suppose a square S is being inserted. The total time spent in computing ρ_u at all nodes $u \in \Lambda(S) \cup \Sigma(S)$ is $O(\log n)$. By Lemma 8, which is given in Section 2.3, the amortized cost of reporting ΔV_u and updating the secondary structures at all nodes in $\Xi(S)$ is $O(\log^2 n + |\Delta V_u|)$. Summing at all nodes of $\Xi(S)$, the total amortized time spent in reporting ΔV is $O(\log^3 n + |\Delta V|)$. Finally, we spend $O(\log n)$ time at each node $u \in \Lambda(S)$ to insert S into $\mathcal{L}_u^*$ and $O(\log^2 n)$ time to insert S into the secondary structures at each node in $\Sigma(S)$. Summing this cost over all nodes in $\Lambda(S) \cup \Sigma(S)$ and adding the cost of credits assigned to S, the total amortized time spent in inserting S is $O(\log^3 n + |\Delta V|)$ time. A similar analysis shows that the amortized time spent in deleting a square is $O(\log^3 n + |\Delta V|)$. Hence, we obtain the following:

▶ **Lemma 6.** *The amortized cost of inserting or deleting a square in* $\mathbb{T}$ *and reporting* ΔV *is* $O(\log^3 n + |\Delta V|)$.

2.3 Reporting ΔV_u via Sweep-Line

We describe the sweep-line procedure for the insertion of a square S; the deletion case is symmetric. Without loss of generality, assume that S contains the top edge of $\square_u$; the procedures for the other cases are similar. Let $\rho_u := \mathrm{cl}(\square_u \setminus \mathcal{U}(\mathcal{L}_u))$ and $S_u^{\mathsf{exp}} := S \cap \rho_u$ as defined above. If $S_u^{\mathsf{exp}} = \varnothing$, there is nothing to do. So assume $S_u^{\mathsf{exp}} \neq \varnothing$. Suppose S_u^{exp} is of the form $\gamma \times [a^-, a^+]$, where γ is the x-span of S_u^{exp} and $a^- < a^+$ are the y-coordinates of the bottom and top edges of S_u^{exp}; the bottom edge of S_u^{exp} is the bottom edge of S, ρ_u, or $\square_u$.

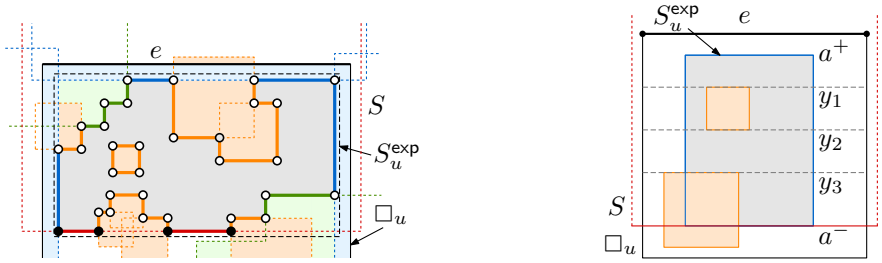

Figure 5 (left) A zoomed-in example of $S_u^{\exp}$ (slightly enlarged for visibility) and the vertices of ΔV where the bottom edge of $S_u^{\exp}$ lies on the bottom edge of the inserted square S. $\mathcal{U}(\mathcal{L}_u)$ is blue, $\mathcal{U}(\mathcal{F}_u) \setminus \mathcal{U}(\mathcal{L}_u)$ is orange, $\mathcal{U}(\mathcal{R}_u) \setminus \mathcal{U}(\mathcal{L}_u \cup \mathcal{F}_u)$ is green, and the edges of S are red. The edges of $\mathcal{U} \cap S_u^{\exp}$ are thick, the old vertices covered by S are hollow, and the new vertices on ∂S are solid. (right) Dashed lines supporting floater edges (orange) with y-coordinates in (a^-, a^+) partition $S_u^{\exp}$ (blue) into rectangles.

Let $V_u^{\text{new}} \subseteq \Delta V_u$ be the set of vertices that are created by the insertion of S (which appear on ∂S) and let $V_u^{\text{old}} \subseteq \Delta V_u$ be the set of vertices that no longer appear on $\mathcal{U}$ after the insertion of S (which lie in $\text{int}(S)$). The vertices of V_u^{new} lie on $S_u^{\exp} \cap \partial S$, i.e., on the bottom edge of $S_u^{\exp}$ if it is contained in the bottom edge of S, and $V_u^{\text{new}} = \varnothing$ if the bottom edge of $\square_u$ is not contained in that of S. Next, a vertex v of V_u^{old} can be classified into the following categories depending on the types of the two (not necessarily distinct) squares S_1, S_2 whose edges contain v: v is a *long-long* (LL) vertex if both S_1 and S_2 are long, in which case v is a vertex of $S_u^{\exp}$ not contained in $\mathcal{U}(\mathcal{S}_u)$, a *long-short* (LS) vertex if S_1 is long and S_2 is short, in which case v is an intersection point of an edge of $\mathcal{U}(\mathcal{S}_u)$ with an edge of $S_u^{\exp}$, and a *short-short* (SS) vertex if both S_1 and S_2 are short, in which case v is a vertex of $\mathcal{U}(\mathcal{S}_u)$ that lies in $\text{int}(S_u^{\exp})$. SS vertices are further classified into three categories, CC, CF, or FF, depending on whether S_1 and S_2 are corners (C) or floaters (F). See Figure 5 (left).

With this characterization of ΔV_u at hand, we report ΔV_u by sweeping downward ($(-y)$-direction) with a horizontal line L_e from $y = a^+$ to $y = a^-$. Let $\mathsf{Y} = \langle y_0 = a^+, y_1, y_2, \ldots, y_t = a^- \rangle$ where $y_1, \ldots, y_{t-1}$ are the y-coordinates of edges of floaters $F \in \mathcal{F}_u$ in the interval (a^-, a^+), sorted in decreasing order. Y can be constructed from $\mathcal{E}_u^y$. The lines $y = y_i$, $0 < i < t$, partition $S_u^{\exp}$ into rectangles. See Figure 5 (right).

The sweep line starts at $y = y_0$ and stops at every y_i, for $0 \le i \le t$. At each y_i, we perform two steps:

(i) Report all vertices of ΔV_u that lie on the line $\mathsf{L}_i : y = y_i$.

(ii) For $i \ge 1$, we report the vertices of ΔV_u that lie in the semi-open rectangle $\sigma_i := \gamma \times (y_i, y_{i-1})$, i.e., all vertices that lie in σ_i but not on its x-edges, using the secondary structures.

We now describe the details of the sweep-line algorithm. As we sweep L_e from a^+ to a^-, we vary ℓ_e, the value associated with the top edge e, to the current position of the sweep-line, so we update the set $\mathcal{I}_e$ (and the secondary structures $\mathbb{W}_e$ and $\mathbb{C}_e$ that store it) as ℓ_e changes. We note that $\mathcal{I}_e$ does not change in the interval (y_i, y_{i-1}). However, when we initialize L_e to y_0, we need to reset ℓ_e to y_0 and update $\mathcal{I}_e$, $\mathbb{W}_e$, and $\mathbb{C}_e$. We will describe this initialization step later and for now assume that $\mathcal{I}_e, \mathbb{W}_e$, and $\mathbb{C}_e$ are consistent with $\ell_e = y_0$. At each y_i, we perform steps (i) and (ii) follows.

Performing step (i). For $i = 0, t$, we set $\delta := \gamma \times \{y_i\}$ to be the top (or bottom) edge of $S_u^{\exp}$. By definition $\delta \cap \text{int}(\mathcal{U}(\mathcal{L}_u)) = \varnothing$. Next, by calling Report-Hole(δ), we compute

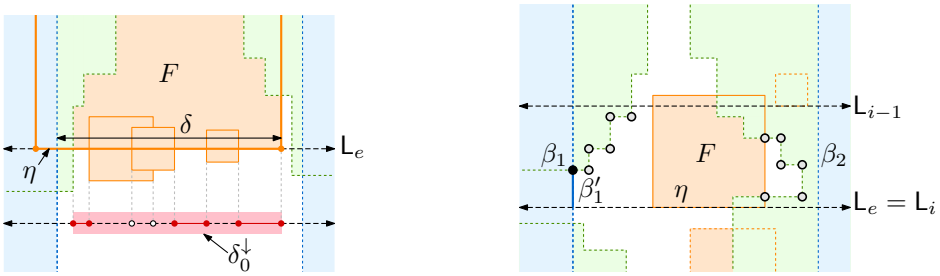

Figure 6 Zoomed-in illustrations of steps (i) and (ii) where the sweep-line L_e reaches the bottom edge η of floater F. Floaters are shown in orange, $\mathcal{U}(\mathcal{L}_u)$ is shown in blue, and $\mathcal{U}(\mathcal{R}_u) \setminus \mathcal{U}(\mathcal{L}_u)$ is shown in green. (left) Step i: The endpoints of the solid intervals in $\delta_0^{\downarrow}$ are the x-projections of the vertices of $\Delta V_u \cap \mathsf{L}_e$. (right) Step ii: The SS vertices in σ_i are hollow, and the lone LS vertex is solid. β_2, the right edge of σ_i, is contained in $\mathcal{U}(\mathcal{R}_u)$ so $\beta_2' = \varnothing$ and it contains no LS vertices.

$\delta_0 := \delta \setminus \text{int}(\mathcal{U}(\mathcal{R}_u)) = \delta \setminus \text{int}(\mathcal{U}(\mathcal{L}_u \cup \mathcal{R}_u))$. Finally, set $\delta_0^{\downarrow}$ to be the x-projection of δ_0. By querying the wall data structure $\mathbb{W}_e$ for edge e with REPORT-HOLES($\delta_0^{\downarrow}$), we compute the intervals of $\delta_0^{\downarrow} \setminus \mathcal{U}(\mathcal{I}_e)$. Let $x_0, \ldots, x_s$ be the endpoints of these intervals. Then, for $0 < j < s$, (x_j, y_i) is a LS vertex. If (x_0, y_i),(x_s, y_i) are endpoints of δ then they are LL vertices, otherwise they are LS vertices.

For $0 < i < t$, L_i contains an x-edge η of a floater $F \in \mathcal{F}_u$. Let $\delta := \eta \cap S_u^{\text{exp}}$. As above, by calling REPORT-HOLE(δ) on the corner data structure $\mathbb{C}_e$ for edge e, we compute $\delta_0 := \delta \setminus \text{int}(\mathcal{U}(\mathcal{L}_u \cup \mathcal{R}_u))$. If η is the bottom edge of F then we first delete the interval $\eta^{\downarrow}$, the x-projection of η, from $\mathbb{W}_e$. Next, we query $\mathbb{W}_e$ with REPORT-HOLES($\delta_0^{\downarrow}$) to report the endpoints of the intervals of $\delta_0^{\downarrow} \setminus \text{int}(\mathcal{U}(\mathcal{I}_e))$. If $x_0, \ldots, x_s$ are these endpoints then (x_j, y_i), for $0 \le j \le s$, are vertices of ΔV_u lying on L_i: If x_0 or x_s lies on $\partial S_u^{\text{exp}}$ then it is an LS vertex, and if it lies on an edge of a square in $\mathcal{R}_u$, (i.e., an endpoint of δ_0 lying inside S_u^{exp}), it is a CF vertex. All other vertices are FF vertices. See Figure 6 (left).

Performing step (ii). Our goal is to report all vertices of $V_u \cap \int(\sigma_i)$. A vertex of ΔV_u not lying on L_{i-1} or L_i lies on an x-edge of a corner square. Since no x-edge of any floater square lies in the interval (y_i, y_{i-1}), the set $\mathcal{I}_e$, and thus $\mathcal{W}_e$, remains the same for all y-values in this interval. Furthermore, $V(\mathcal{R}_u \cup \mathcal{F}_u) \cap \sigma_i = V(\mathcal{R}_u \cap \mathcal{W}_e) \cap \sigma_i$. We can thus report all CF and CC vertices lying in σ_i by querying $\mathcal{R}_e$ with REPORT-VERTICES(σ_i). The $O(1)$ LS vertices in σ_i are defined by a long square, namely a long square that defines a y-edge of σ_i, and a corner square. For each y-edge β_j of σ_i, we call $\mathbb{W}_e$ with REPORT-HOLE($\beta_j^{\downarrow}$). If $\beta_j^{\downarrow}$ is returned, then β_j is not contained in $\mathcal{U}(\mathcal{W}_e) \cap \sigma_i = \mathcal{U}(\mathcal{F}_u) \cap \sigma_i$ and we call $\mathbb{C}_e$ with REPORT-HOLE(β_j); let $\beta_j' \subseteq \beta_j$ be the returned (possibly empty) interval. The endpoints of β_j^* that lie in int(β_j), if any, are the LS vertices on β_j. See Figure 6 (right). Finally, for $i < t$, if η is the top edge of F, we insert $\eta^{\downarrow}$ into $\mathbb{W}_e$ and $\mathbb{C}_e$, otherwise η is the bottom edge of F and we delete $\eta^{\downarrow}$ from $\mathbb{C}_e$. (In the latter case, $\eta^{\downarrow}$ was already deleted from $\mathbb{W}_e$ in the previous step.) Note that if η is an edge of a floater F, then $\eta^{\downarrow}$ is accordingly inserted or deleted to $\mathbb{W}_e$ and $\mathbb{C}_e$ by the end of these two steps, and hence they are made consistent with ℓ_e.

When the sweep procedure ends, we set $\ell_e := a^-$. Note that now $\mathcal{I}_e$ consists of the x-projections of floaters that intersect the line $y = a^-$, as desired. Therefore the secondary structures $\mathbb{W}_e$ and $\mathbb{C}_e$ are consistent with the new value of ℓ_e.

Initializing the sweep line. At the beginning of the procedure, the value of ℓ_e is the value at which the sweep procedure stopped after inserting or deleting a long square at u that contained the top edge of $\square_u$. To initialize the sweep line at $y = a^+$, the top edge of S_u^{exp}, and to initialize $\mathbb{W}_e$ and $\mathbb{C}_e$ correctly for $y = a^+$, we again perform a line sweep from the current value of ℓ_e to a^+ by a horizontal line L_e as above, except that no reporting of vertices occurs. That is, as we sweep, we stop at each encountered x-edge η of a floater square, and insert or delete the interval $\eta^{\downarrow}$ in $\mathbb{W}_e$ and $\mathbb{C}_e$ without (querying for and) reporting any vertices.

Runtime analysis. Let κ_u be the number of vertices in ΔV_u, and let a_{pr} be the value of ℓ_e before the sweep line is initialized. For the insertion of a square S, let φ_u be the number of y-coordinates of floater edges in the intervals $[a^-, a^+]$ and $[\ell_e, a^+]$ (resp. $[a^+, a_{pr}]$) if $a_{pr} \leq a^+$ (resp. $a_{pr} > a^+$). For the deletion of a square S, let φ_u be the number of y-coordinates of floater edges in the intervals $[a^-, a^+]$ and $[\ell_e, a^-]$ (resp. $[a^-, a_{pr}]$) if $a_{pr} \leq a^-$ (resp. $a_{pr} > a^-$). (Some y-coordinates may be counted twice by φ_u.) Thus, the total time spent for the insertion or deletion of S at each node $u \in \Xi(S)$ is $(1 + \varphi_u) \cdot O(\log^2 n) + O(\kappa_u)$.

We charge $O(\log^2 n)$ units of time to each of the φ_u floater edges that are parallel to the sweep line and that were crossed by the two sweep-line procedures — one in the initialization step and one for reporting the vertices. This charging pays for the $\varphi_u \log^2 n$ term in the running time. The amortized cost of reporting ΔV_u at u is $O(\log^2 n + \kappa_u)$, provided that each floater at u had enough credits to pay for the costs charged to it. The following lemma proves that the floater is not charged too many times.

▶ **Lemma 7.** *Let f be an edge of a floater F at a node $u \in \mathbb{T}$. Then f is charged by the sweeps at $\square_u$ at most six times during the entire algorithm.*

Proof. Without loss of generality assume that f is an x-edge. Then f is charged by the *top* and *bottom* sweeps, i.e., sweeps performed when a square $S \in \mathcal{L}_u^*$ containing the top or the bottom edge of $\square_u$ is inserted or deleted. We claim that f is charged at most three times by the top sweep — a similar argument holds for the bottom sweep. Let e denote the top edge of $\square_u$. Recall that f is charged whenever the horizontal sweep-line L_e crosses f — either in the initialization step or in the reporting step. Furthermore L_e will cross f from opposite directions, i.e., when L_e is sweeping *downward* ($(-y)$-direction) or *upward* ($(+y)$-direction), in any two consecutive crossings. We claim that L_e crosses f in at most three top sweeps.

Recall that during the insertion or deletion of a square S, either $S_u^{\mathsf{exp}} = \varnothing$ and no sweeps occur, otherwise $S_u^{\mathsf{exp}} \neq \varnothing$ and two sweeps occur (one in the initialization step and one in the reporting step). We classify downward (resp. upward) sweeps that occur during an insertion of a square as DI (resp. UI) sweeps, and classify downward (resp. upward) sweeps that occur during a deletion of a square as DD (resp. UD) sweeps. Consider the sequence of sweeps that cross f while $F \in \mathcal{F}_u$, i.e., after the insertion of F and before the deletion of F at node u. We claim the sequence satisfies the following two constraints:

(i) No sweeps can occur after a DI sweep.
(ii) No sweeps can occur before a DD sweep.

Clearly the longest valid sequences of sweeps are of the form "DD, UD, DI" or "DD, UI, DI," and hence f is crossed by L_e in at most three top sweeps. (It can be shown that the latter sequence is not possible, but this fact is not needed in order to prove the lemma.) It remains to prove the constraints above.

Proof of claim (i). Suppose L_e crosses f in a DI sweep, and let $S \in \mathcal{L}_{u,e}^*$ be the square being inserted. We argue that if the bottom edge of g of S lies above f then f could not

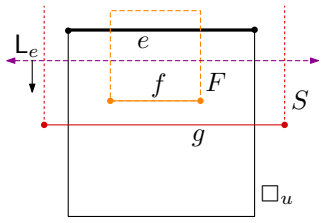

Figure 7 An illustration of the proof of Lemma 7: Sweep-line L_e is swept downward toward edge f of floater F during the insertion or deletion of S at node u.

have been crossed in this sweep, either during the initialization step or during the reporting step, as follows. $S_u^{\mathsf{exp}} \neq \varnothing$, so the bottom edge of $S_u^{\mathsf{exp}} = S \cap \rho_u$ is either contained in g or lies above it. A downward sweep in the initialization step sweeps to the top edge of S_u^{exp}, and a downward sweep in the reporting step sweeps from the top edge of S_u^{exp} to the bottom edge of S_u^{exp}; in either case, the sweeps stop above g and hence above f, so f is not crossed. So we assume that g lies below f; see Figure 7. Note that an upward sweep, either in the initialization step of the insertion of a square or in the reporting step of the deletion of a square, always stops at the top edge of ρ_u. After S has been inserted into $\mathcal{L}^*_{u,e}$ and until it is deleted from $\mathcal{L}^*_{u,e}$, the top edge of ρ_u is contained in g or lies below it, which implies that no upward sweep will cross g until after S is deleted. Let $|B|$ denote the length of the longest side of any rectangle B. Since F is a floater at u and S contains an edge of $\square_u$ (as S is long at u), $|F| < |\square_u| < |S|$. Then, since S is inserted after F, S will be deleted after F has been deleted by property (P1), which implies that after L_e crosses f during the insertion of S, f will not be crossed by L_e in any subsequent sweeps. This proves claim (i).

Proof of claim (ii). Suppose L_e crosses f in a DD sweep, where $S \in \mathcal{L}^*_{u,e}$ is the square being deleted. This sweep occurs in the initialization step of the deletion of S as L_e moves from some position above f to the bottom edge of S_u^{exp}, which must be below f. The latter edge lies above the bottom edge g of S. See Figure 7 again. As noted above, in an upward sweep, either in the initialization step of the insertion of a square or in the reporting step of the deletion of a square, always stops at the top edge of ρ_u. While S is present, the top edge of ρ_u is contained in g or lies below it, which implies that no upward sweep can cross g until after S is deleted. Again, we have that $|F| < |\square_u| < |S|$, and hence property (P1) implies S is inserted before F since f is present during the deletion of S. Since g is below f, no upward sweeps preceding the deletion of S can cross f, proving claim (ii). ◀

Lemma 7 implies that each floater edge in $\mathcal{F}_u$ has sufficient credits to pay for the cost charged to it. Hence, we obtain the following:

▶ **Lemma 8.** *The amortized cost of inserting/deleting a square S at a node $u \in \Xi(S)$ is $O(\log^2 n + |\Delta V_u|)$ and at a node $u \in \Sigma(S) \setminus \Xi(S)$ is $O(\log^2 n)$.*

▶ Remark 9. The sweep-line algorithm and secondary data structures can be extended so that not only they compute ΔV_u, but also compute (the xy-projection of) $\partial\mathcal{U}$ within $S \cap \square_u$ in the same time bound. Hence, we can compute $\partial\mathcal{U}$ within each rectangle of $\Pi(S)$. By merging these pieces together over all rectangles of $\Pi(S)$, we can compute (the xy-projection of) $\partial\mathcal{U}$ within S. We thus compute $\partial\mathcal{U}$ on each horizontal face of the cubes in $\mathcal{C}$ in time $O(n \log^3 n + \kappa)$. By performing the plane-sweep in the x-direction and y-direction, we can compute $\partial\mathcal{U}$ on the other faces of cubes in $\mathcal{C}$ as well. These sweeps will be simpler because we already have computed the vertices of $\mathcal{U}$. We omit the details from this version.

2.4 Secondary structures

In this section, we describe the details of the wall and corner data structures at every node $u \in \mathbb{T}$ and edge e of $\square_u$.

Wall data structure. For $\mathbb{W}_e$, we use the data structure described by Wood [18], which is a standard segment tree augmented with additional information stored at each of its nodes. It supports our desired operations, Insert, Delete, Membership, and Report-Holes in the time mentioned earlier. Its size is $O(|\mathsf{X}_u|)$ and is constructed in $O(|\mathsf{X}_u|)$ time at the preprocessing stage of the algorithm. Every point $p \in \mathsf{X}$ is contained in region $\square_u$ of exactly one node u in each level of $\mathbb{T}$. $\mathbb{T}$ has $O(\log n)$ levels, so constructing all wall data structures takes $O(\sum_u |\mathsf{X}_u|) = O(n \log n)$ time.

Corner data structure. For each edge e of $\square_u$, we construct the data structure of Agarwal [1, Lemma 5]. Although it was originally described to support *area* queries (report the area of $\rho \cap \mathcal{U}(\mathcal{R}_u) \setminus \mathcal{U}(\mathcal{W}_e)$ for a given query rectangle ρ), it is straightforward to extend the data structure to support our required operations, Report-Hole and Report-Vertices. In particular, to answer the area queries, it maintains the x-edges (resp. y-edges) of $\mathcal{U}(\mathcal{R}_u \cup \mathcal{W}_e)$ that lie on x-edges (resp. y-edges) of corner squares sorted by their y-coordinates (resp. x-coordinates), which is sufficient to answer our desired operations by searching over these lists of edges. Like $\mathbb{W}_e$, it is constructed in $O(|\mathsf{X}_u|)$ time at the preprocessing stage of the algorithm, so constructing all corner data structures takes $O(\sum_u |\mathsf{X}_u|) = O(n \log n)$ time.

3 Algorithm for the Unbounded-Spread Case

In this section, we extend the algorithm of Section 2 to the case when the spread of X, the xy-projections of the vertices of $\mathcal{C}$, is arbitrary. The algorithm is largely the same. The main challenge is that we cannot use a standard quadtree for our dynamic data structure $\mathbb{T}$, as it may have $\Omega(n)$ depth even if we use a compressed quadtree. Instead, $\mathbb{T}$ is a compressed quadtree with fingers[2] [13]. Its height is $O(\log n)$ regardless of the spread of X, its size is $O(n)$, and it can be constructed in $O(n \log n)$ time.

The properties of $\mathbb{T}$ are as follows. Every node $u \in \mathbb{T}$ is associated with a region $\boxdot_u$ that is a square or the difference of two nested squares $\square_u^O \setminus \square_u^I$, where $\square_u^O \supset \square_u^I$. For nodes u at which $\boxdot_u$ is a square, we say $\square_u^O = \boxdot_u$ and $\square_u^I = \varnothing$ so that $\square_u^O, \square_u^I$ are well-defined for all nodes of $\mathbb{T}$. For the root node r, $\boxdot_r \supset \mathsf{X}$. If $|\mathsf{X} \cap \boxdot_u| \leq 1$, u is a leaf; otherwise u is an internal node and $\boxdot_u$ is partitioned either into four regions with congruent outer squares, or it is partitioned into $\boxdot_u \setminus \square_u^S$ and $\square_u^S$ by a square $\square_u^S$ such that $\square_u^O \supset \square_u^S \supset \square_u^I$. In either case, the regions in the partition of $\boxdot_u$ are associated with a child of u, and hence u has either two or four children. See Figure 8 (left). Recall that $|\rho|$ denotes the length of the longest side length of any rectangle ρ. Any nodes where $\square_u^I \neq \varnothing$, $\mathbb{T}$ has the property that $\square_u^I$ is *sticky*; that is, the distance between the top (resp. right, bottom, left) edge of $\square_u^O$ and the top (resp. right, bottom, left) edge of $\square_u^I$ is either 0, in which case the former contains

[2] The regions associated with the nodes of a compressed quadtree with fingers as described in [13] may have multiple holes, whereas the *BBD trees* proposed in [5] have at most one hole per node but rectangular regions (which may not be squares). By combining some of the ideas from the construction of BBD trees in [5] to ensure each region has at most one hole, our tree $\mathbb{T}$ can be constructed with all stated properties.

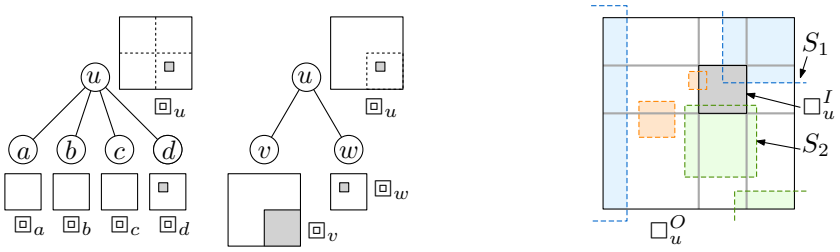

Figure 8 (left) Two examples of internal nodes in $\mathbb{T}$ with identical regions $\boxdot_u$. On the left, the outer square of $\boxdot_u$ is partitioned into four congruent squares, and on the right, $\boxdot_u$ is partitioned by a quadrant of its outer box. (right) Examples of long (blue), floater (orange), and corner (green) squares at rectangles of ∇_u. S_1 (resp. S_2) is a long (resp. corner) square for each rectangle of ∇_u that it intersects (even though S_1 is not a long square for $\square_u^O$).

the latter, or at least $|\square_u^I|$. For example, if $[o_1, o_2]$ (resp. $[i_1, i_2]$) is the x-projection of $\square_u^O$ (resp. $\square_i^I$), then for $j = 1, 2$ we have that $|o_j - i_j|$ is 0 or at least $|i_1 - i_2|$.

For a node u, we define long and short squares and the sets $\mathcal{S}_u$, $\mathcal{L}_u$, and $\mathcal{L}_u^*$ as earlier. Note that a vertex of a square $S \in \mathcal{L}_u$ may lie in the inner square $\square_u^I$ of $\boxdot_u$ (in fact, at most one vertex of S); see Figure 8 (right). For a square S, let $\Lambda(S)$, $\Sigma(S)$, $\Xi(S)$, and $\Pi(S)$ be the same as before. We report $\Delta V_u := \Delta V \cap \square_u$ for each node $u \in \Xi(S)$. However, because $\boxdot_u$ may have a hole, reporting ΔV_u at u is more involved than a single sweep through $\boxdot_u$. We describe what we store at each node u, then how we update $\mathbb{T}$ and ultimately report ΔV_u at each node $u \in \Xi(S)$ due to the insertion or deletion of a square S at u.

The information at node u. We avoid all issues involving the inner square $\square_u^I$ by further partitioning $\boxdot_u$ into rectangles, then using essentially the same algorithm. In particular, the lines supporting the inner square $\square_u^I$ (if it exists) induce a partition of $\square_u$ into a set ∇_u of $m \leq 8$ non-empty rectangles $\{B_u^1, \ldots, B_u^m\}$; if $\square_u^I = \varnothing$, set $\nabla_u := \{\boxdot_u\}$. See Figure 8 (right). For a rectangle $B \in \nabla_u$, let $\mathcal{L}_B$ (resp. $\mathcal{S}_B$) be the set of long (resp. short) squares at u that intersect B, and let $\mathcal{L}_B^* := \mathcal{L}_B \cap \mathcal{L}_u^*$. Let $\mathcal{F}_B$ be the subset of short squares with at least two vertices in $\text{int}(B)$, and let $\mathcal{R}_B := \mathcal{S}_B \setminus \mathcal{F}_B$ be the set of short squares with at most one vertex in $\text{int}(B)$. The following lemma will be crucial for our runtime analysis.

▶ **Lemma 10.** *For any node u of $\mathbb{T}$, rectangle $B \in \nabla$, and square $S \in \mathcal{L}_B$, $|S| > |B|$ and hence S contains an edge of B.*

Proof. Let S be a square in $\mathcal{L}_B$. S intersects $\text{int}(\boxdot_u)$ but no vertex of S lies in $\text{int}(\boxdot_u)$, so its vertices either lie in $\text{int}(\square_u^I)$ or outside $\square_u^O$. There are two cases. First, suppose all vertices of S lie outside $\square_u^O$. Then $|S| > |\square_u^O| > |B|$, as desired.

Next, suppose a vertex v_1 of S lies in $\text{int}(\square_u^I)$. If $S \supset B$, we are done, so suppose otherwise. No vertices of S lie in $\text{int}(B)$, so an edge $e := v_1v_2$ of S spans B where v_2 is a vertex of S that lies outside $\square_u^O$. (If all vertices of S lie inside $\text{int}(\square_u^I)$ then S does not intersect B.) See Figure 9. By construction of ∇_u, the edges of B perpendicular to e have length $|\square_u^I|$ (in particular, one is an edge of $\square_u^I$ and the other is a portion of an edge of $\square_u^O$). By the sticky property of $\square_u^I$, the edges of B parallel to e has length at least $|\square_u^I|$, and thus at least as long as the former ones. Since e is longer than the edges of B parallel to it, $|S| > |B|$. ◀

To be consistent with the previous algorithm, we refer to the squares in $\mathcal{F}_B$ as *floater squares* and the squares in $\mathcal{R}_B$ as *corner squares*, even though a square $S \in \mathcal{R}_B$ may not

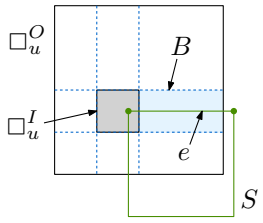

Figure 9 An illustration of the proof of Lemma 10: An edge e of a long square (green) intersecting rectangle $B \in \nabla_u$.

have any vertices in $\text{int}(B)$ and may even contain B. For such a corner square $S \in \mathcal{R}_B$, we associate S with the top-left vertex v_1 of B if $v_1 \in S$, otherwise we associate S with the bottom-right vertex v_2 of B (in which case $v_2 \in S$).

We maintain $\mathcal{L}_B^*$ and $\mathcal{S}_B$ for each $B \in \nabla_u$ in the same fashion as $\mathcal{L}_u^*, \mathcal{S}_u$ were stored at nodes u of the quadtree in the previous algorithm. Note that by Lemma 10, we have the property that any square $S \in \mathcal{L}_B$ contains an edge of B as before. For each edge e of B, we store the long squares that contain e, $\mathcal{L}_{B,e}^*$, in red-black trees as before. We similarly store the x-edges (resp. y-edges) of floater squares in $\mathcal{F}_B$ sorted by their y-coordinates (resp. x-coordinates), which we call $\mathcal{E}_B^x$ (resp. $\mathcal{E}_B^y$).

For each edge e of a rectangle $B \in \nabla_u$, we maintain a value ℓ_e that is the position of the sweep-line L_e associated with edge e, a wall data structure $\mathbb{W}_e(\mathcal{I}_e)$ and corner data structure $\mathbb{C}_e(\mathcal{R}_B, \mathcal{I}_e)$, where $\mathcal{I}_e$ are the x-projections (resp. y-projections) of squares in $\mathcal{F}_B$ intersected by L_e if e is an x-edge (resp. y-edge). Recall their descriptions from Section 2.2.

Reporting ΔV_u. To report ΔV_u when we insert (or delete) a square S, we report $\Delta V_B := B \cap \Delta V_u$ for each rectangle $B \in \nabla_u$ intersected by S (there are no vertices of ΔV_B if $S \cap B = \varnothing$). The insertion and deletion procedures are largely the same as before. In particular, we employ the sweep-line approach from the previous algorithm and essentially treat B as if it was $\square_u$ to report ΔV_B, as follows.

3.1 Reporting ΔV

We describe the procedure to report ΔV and update the data structures for the insertion of a square S; the procedure for the deletion of S is a similar extension of the deletion procedure described in Section 2.2 for bounded spread. There are four main steps:

(1) At each node $u \in \Lambda(S) \cup \Sigma(S)$, we compute $\rho_B := \text{cl}(B \setminus \mathcal{U}(\mathcal{L}_B))$ for all rectangles $B \in \nabla_u$ using a top-down traversal of $\mathbb{T}$ and the sequences of $\mathcal{L}_B^*$, as described below.

(2) At each node $u \in \Xi(S)$, we report ΔV_B for all rectangles $B \in \nabla_u$ by sweeping a line from an edge of rectangle $S_B^{\mathsf{exp}} := S \cap \rho_B$, i.e., the portion of $S \cap B$ that is left "exposed" by the squares of $\mathcal{L}_B$.

(3) For each node $u \in \Lambda(S)$, we insert S to $\mathcal{L}_B^*$ for each rectangle $B \in \nabla_u$ that S intersects. In particular, if S contains the edge e of B, S is inserted into $\mathcal{L}_{B,e}^*$; recall that if $B \subseteq S$, then S is associated with the top edge of B.

(4) For each rectangle $B \in \nabla_u$ at node $u \in \Sigma(S)$, we update $\mathcal{S}_B$ and its secondary structures, as follows. If $S \cap B = \varnothing$, there is nothing to do, so suppose otherwise. If S is a corner square for B, we insert S into all four corner data structures $\mathbb{C}_e$ stored at B for each of its edges. Otherwise, S is a floater square, and we first insert the x-edges (resp. y-edges) of S which intersect B into $\mathcal{E}_B^x$ (resp. $\mathcal{E}_B^y$). Then, for each x-edge (resp. y-edge) e of

B such that ℓ_e lies in the y-span (resp. x-span) of S, we insert the x-projection (resp. y-projection) of S to $\mathbb{W}_e$ and $\mathbb{C}_e$.

We now describe the first two steps in more detail. Consider a node $u \in \Lambda(S) \cup \Sigma(S)$, and assume that the rectangle ρ_D has been computed for all $D \in \nabla_{p(u)}$. Let $\rho_B^* \coloneqq B \setminus \mathcal{U}(\mathcal{L}_B^*)$, which can be computed in $O(1)$ time using the sequences of $\mathcal{L}_B^*$. Then ρ_B can be computed in $O(1)$ time using ρ_B^* and the rectangles ρ_D for each $D \in \nabla_{p(u)}$, using the fact that

$$\begin{aligned}\rho_B &= B \setminus \mathcal{U}(\mathcal{L}_B) = B \setminus \mathcal{U}(\mathcal{L}_B^* \cup \mathcal{L}_{p(u)}) = (B \setminus \mathcal{U}(\mathcal{L}_B^*)) \cap (B \setminus \mathcal{U}(\mathcal{L}_{p(u)}) \\ &= \rho_B^* \cap (\boxdot_{p(u)} \setminus \mathcal{U}(\mathcal{L}_{p(u)})) = \rho_B^* \cap \Big(\bigcup_{D \in \nabla_{p(u)}} B_D \setminus \mathcal{U}(\mathcal{L}_D) \Big) \\ &= \rho_B^* \cap \Big(\bigcup_{D \in \nabla_{p(u)}} \rho_D \Big) = \bigcup_{D \in \nabla_{p(u)}} (\rho_B^* \cap \rho_D).\end{aligned}$$

That is, ρ_B is the union of at most eight interior-disjoint rectangles, each of which is of the intersection of rectangles ρ_B^* and ρ_D for a rectangle $D \in \nabla_{p(u)}$, which we assume have been precomputed. Hence, ρ_B can be computed in $O(1)$ time. This completes step (1).

Let B be a rectangle of ∇_u for a node $u \in \Xi(S)$. Note that S_B^{exp} and ρ_B are indeed rectangles, since any square in $\mathcal{L}_u$ contains an edge of B by Lemma 10 and B is a rectangle. The main idea is that even though the sweep-line procedure to report ΔV_u in Section 2.3 was described for a square, the correctness is invariant of whether it is a square or a rectangle as in this scenario. We report ΔV_B in the same way as before: If $u \in \Sigma(S)$, then ΔV_B is the set of $O(1)$ vertices of $S \cap \rho_B$ that lie in $\operatorname{int}(S \cap B)$ and hence can be computed in $O(1)$ time. Otherwise, $u \in \Lambda(S)$. Suppose S contains the top edge e of B. If $S_B^{\mathsf{exp}} = \varnothing$, $\Delta V_B = \varnothing$, and there is nothing to. So assume $S_B^{\mathsf{exp}} \neq \varnothing$. We first initialize the sweep line L_e to be at the top edge of rectangle S_B^{exp}, then we sweep it to its bottom edge and report all vertices of ΔV_B using $\mathbb{W}_e$ and $\mathbb{C}_e$ in the process. The details of the sweep-line procedure are the same as in the bounded spread case; see Section 2.3.

Runtime analysis. Most of the runtime analysis of the previous algorithm easily extends to this algorithm. Let S be a square being inserted or deleted. Since $|\nabla_u| \leq 8$ for each node $u \in \mathbb{T}$, inserting S to the secondary structures at each rectangle $B \in \nabla_u$ of a node u in $\Lambda(S)$ (resp. $\Sigma(S)$) still takes $O(\log n)$ (resp. $O(\log^2 n)$) time. To extend the amortized time to report ΔV_B for a rectangle $B \in \nabla_u$ of a node $u \in \Xi(S)$, we again charge $O(\log^2 n)$ time to the edges of floaters encountered during the sweep-line procedure at B. Then, as in Lemma 7, it can be shown that each such edge is charged at most six times by the sweeps at B during the entire algorithm. However, there is a subtle issue when extending the proof of the lemma to this setting: the proof used the inequalities $|S| > |B| > |F|$ for any floater $F \in \mathcal{F}_B$ in order to conclude $|S| > |F|$ and apply property (P1), but our justification for those inequalities relied on B being a square in that setting. We instead use Lemma 10 to conclude $|S| > |F|$, which crucially relies on the sticky property of $\mathbb{T}$.

The remainder of the analysis follows as before: it follows that the amortized runtime to perform the at most eight sweeps (at most one per rectangle of ∇_u) at node $u \in \Xi(S)$ is $O(\log^2 n + |\Delta V_u|)$, and hence inserting or deleting S to $\mathbb{T}$ and reporting ΔV takes $O(\log^3 n + |\Delta V|)$ time. Thus, the entire algorithm takes $O(n \log^3 n + \kappa)$ time, where κ is the number of vertices of $\mathcal{U}$, proving Theorem 1.

4 Algorithms for Special Cases

In this section, we simplify the algorithm for two special cases in which $\mathcal{C} \coloneqq \{C_1, \ldots, C_n\}$ is a set of n axis-aligned cubes in $\mathbb{R}^3$ in general position that:

1. have bounded *depth*, i.e., the maximum number of cubes in $\mathcal{C}$ that contain any point $p \in \mathbb{R}^3$ is bounded by a constant $c > 0$, or
2. are all congruent.

4.1 Cubes with Bounded Depth

We assume that the depth of the cubes in $\mathcal{C}$ is bounded by a constant. For simplicity, we also assume that the cubes in $\mathcal{C}$ have bounded spread; it is straightforward but slightly more tedious to extend the following techniques to the unbounded spread case. In this case, we employ the same algorithm as described in Section 2, except that we no longer need the corner data structures and can perform their operations in $O(1)$ time, as explained below. As a result, the amortized runtime to report ΔV due to the insertion or deletion of a square in $\mathcal{S}$ improves to $O(\log^2 n + |\Delta V|)$, as follows.

Fix a rectangle $B \in \nabla_u$ for a node $u \in \mathbb{T}$, and let $v \coloneqq (x_i, y_i)$ be one of the four vertices of B. For any $a \in \mathbb{R}$, $O(1)$ cubes of $\mathcal{C}$ contain (x_i, y_i, a), and hence $O(1)$ squares in $\mathcal{S}$ contain v at any point during the plane-sweep. In particular, $O(1)$ squares contain any of the four vertices of B. Any square in $\mathcal{R}_B$ contains a vertex of B, and hence $|\mathcal{R}_B| = O(1)$. Then $\mathcal{U}(\mathcal{R}_B)$, and thus $\mathcal{U}(\mathcal{R}_B) \cap \square_u$, has constant complexity, so the latter can be maintained explicitly in $O(1)$ time per insertion or deletion to $\mathcal{R}_u$ (at worst, it takes $O(1)$ time to recompute it from scratch any time that it is needed). Furthermore, given any query segment $\sigma \supset B$, $\sigma \setminus \operatorname{int}(\mathcal{U}(\mathcal{R}_u))$ can be computed in $O(1)$ time, which replaces the need for the Report-Hole(σ) operation of the corner data structures.

Lastly, we need to implement the Report-Vertices(ρ) operation of the corner data structures for each edge e of B using $\mathcal{U}(\mathcal{R}_u) \cap \square_u$ and $\mathbb{W}_e$, where $\rho \subset \square_u$ is a query rectangle. Suppose e is an x-edge of B. We first compute $\mathcal{U}_\rho \coloneqq \mathcal{U}(\mathcal{R}_u) \cap \rho$ in $O(1)$ time, and then, for each edge σ of $\mathcal{U}_\rho$, we compute the endpoints of the intervals of $\sigma^\downarrow \setminus \operatorname{int}(\mathcal{U}(\mathcal{I}_e))$ in $O(\log n + \kappa_\sigma)$ time, where $\sigma^\downarrow$ is the x-projection of σ and κ_σ is the number of endpoints reported. These 1D vertices correspond to the 2D vertices of $\mathcal{U}(\mathcal{S})$ on σ, and any vertex is reported at most twice. Thus, the overall runtime for this operation is $O(\log n + \kappa)$, where κ is the total number of vertices reported.

Plugging these improved bounds for the corresponding operations of the corner data structures in the analysis of the sweep-line procedure, we have that the amortized time to report ΔV for the insertion or deletion of a square S to $\mathcal{S}$ is $O(\log^2 n + |\Delta V|)$. Given that κ, the total number of vertices of $\mathcal{U}$, is $O(n)$ in this case, the overall runtime is $O(n \log^2 n)$, as claimed in Theorem 3.

4.2 Congruent Cubes

Without loss of generality, assume that all cubes in $\mathcal{C}$ are unit cubes. In this case, $\mathcal{S}$ is now a set of unit squares in $\mathbb{R}^2$. Whenever the sweeping plane reaches the top or bottom face of a cube in $\mathcal{C}$, we neither need a tree data structure nor do we need a 2D sweep-line procedure to report ΔV. Instead we only need a 2D grid and a simpler version of the corner data structure, as described below.

The data structure. Let $\mathcal{G}$ be the 2-dimensional integer grid. Without loss of generality, no point of X lies in on the grid lines of $\mathbb{R}^2$. For $i, j \in \mathbb{Z}$, let $\square_{i,j}$ denote the grid cell $[i, i+1] \times [j, j+1]$. For all $i, j \in \mathbb{Z}$, let $\mathsf{X}_{i,j} \coloneqq \mathsf{X} \cap \square_{i,j}$, and let $\mathcal{G}^*$ be the non-empty grid cells of $\mathcal{G}$, i.e., $\mathcal{G}^* = \{\square_{i,j} \in \mathcal{G} \mid |\mathsf{X}_{i,j}| > 0\}$.

For any square $S \in \mathcal{S}$, any grid cell $\square \in \mathcal{G}^*$ intersected by S contains exactly one vertex of S; that is, S is a *corner square* for $\square$. For any grid cell $\square \in \mathcal{G}^*$, let $\mathcal{S}_\square$ be the set of squares in $\mathcal{S}$ that intersect $\square$. Since there are no long or floater squares at any grid cell $\square \in \mathcal{G}^*$, there are no (projections of) floaters to maintain, nor any vertices of ΔV defined by such squares to report, which accounts for much of the intricacies of the previous algorithms.

For each $\square \in \mathcal{G}^*$, we build one *corner data structure* $\mathbb{C}_\square$, that maintains $\mathcal{S}_\square$ and supports the following operations:

- Insert/Delete(S): Insert or delete a corner square S to $\mathcal{R}_\square$.
- Report-Hole(σ): Given a query axis-aligned segment $\sigma \subset \square$, return the (at most one) interval of $\sigma \setminus \operatorname{int}(\mathcal{U}(\mathcal{S}_\square))$.
- Report-Vertices(ρ): Let $\rho \subseteq \square$ be a corner rectangle, i.e., one of its vertices is a vertex of $\square$. Return $V(\mathcal{S}_\square) \cap \rho$.

As in Section 2.1, we implement $\mathbb{C}_\square$ using the data structure of Agarwal [1, Lemma 5], which supports the operations above without modification. An Insert/Delete takes $O(\log^2 n)$ time, Report-Hole takes $O(\log n)$ time, and Report-Vertices takes $O(\log n + \kappa)$ time, where κ is the number of vertices reported. Note that, in contrast with the corner data structures used in the previous two algorithms, we build only one for each cell $\square$ instead of one per edge of $\square$, and this data structure does not maintain a set of intervals in addition to the set of corner squares $\mathcal{S}_\square$. Using the fact that there are no intervals and the query rectangle is a corner rectangle, the corner data structure in [1] can be simplified, but we skip the details here.

Reporting ΔV. We describe the procedure to report ΔV and update the data structures for the insertion of a square S; the procedure for the deletion of S is a similar extension of the deletion procedure described in Section 2.2 for bounded spread. Let $\Sigma(S)$ be the four grid cells that S intersects. The grid cells in $\Sigma(S)$ partition S. We report $\Delta V_\square \coloneqq \Delta V \cap \square$ for each grid cell $\square \in \Sigma(S)$.

Let $V^{\text{new}} \subseteq \Delta V$ be the set of vertices that are created by the insertion of S (which appear on ∂S), and let V^{old} be the set of vertices that no longer appear on $\mathcal{U}$ after the insertion of S (which lie in $\operatorname{int}(S)$).

Fix a grid cell $\square \in \Sigma(S)$. Let e_1, e_2 be the edges incident to the vertex of S in $\operatorname{int}(\square)$. The vertices of $V^{\text{new}} \cap \square$ that lie on e_1 (resp. e_2) are the endpoints of $e_1 \setminus \operatorname{int}(\mathcal{U}(\mathcal{S}_\square))$ (resp. $e_2 \setminus \operatorname{int}(\mathcal{U}(\mathcal{S}_\square))$ that lie in $\operatorname{int}(\square)$. See Figure 10. There are at most two such vertices lying on each edge e_i, and they are computed by calling $\mathbb{C}_\square$ with Report-Hole($e_i \cap \square$)). Then we report $V^{\text{old}} \cap \square$, i.e., the vertices of $\mathcal{U}(\mathcal{S}_\square) \cap \square$, by calling $\mathbb{C}_\square$ with Report-Vertices($S \cap \square$). The vertices reported account for all vertices of ΔV. Finally, we insert S to $\mathcal{S}_\square$ by calling $\mathbb{C}_\square$ with Insert(S). Repeating this step for all four cells in $\Sigma(S)$, we report ΔV.

The eight Report-Hole calls take $O(\log n)$ time overall, the four Report-Vertices calls take $O(\log n + |\Delta V|)$ time, and the four Insert calls take $O(\log^2 n)$ time. Hence, the total time spent for the insertion of S is $O(\log^2 n + |\Delta V|)$. Similarly, the deletion of a square takes $O(\log^2 n + |\Delta V|)$ time.

Runtime analysis. $\mathcal{G}^*$ and $\Sigma(S)$ for all squares S can be computed in $O(n \log n)$ time. At the beginning of the algorithm, no cubes intersect the sweeping plane and hence $\mathcal{S}_\square = \varnothing$, so

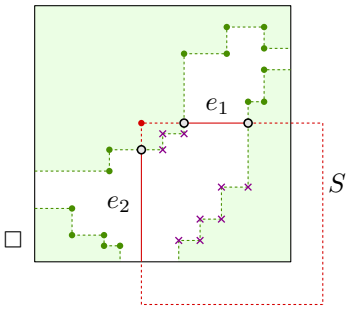

Figure 10 An illustration of a square S (red) being inserted at a grid cell $\square \in \Sigma(S)$ (black). The solid portions of edges e_1,e_2 of S are the portions of the edges returned by the calls to REPORT-HOLE. The vertices of $V^{\text{new}} \cap \square$ are marked as hollow and lie on e_1 and e_2, and the vertices of $V^{\text{old}} \cap \square$ are marked as crosses and lie in $\text{int}(S) \cap \square$.

building $\mathcal{C}_\square$ for $\square \in \mathcal{G}^*$ takes $O(n)$ time overall. Given that κ, the total number of vertices of $\mathcal{U}$, is $O(n)$ in this case, the overall runtime is $O(n \log^2 n)$, as claimed in Theorem 3.

Unit cubes with bounded depth. Suppose the cubes of $\mathcal{C}$ are unit cubes with bounded depth. Let $\mathcal{G}$ be the 3D integer grid, which partitions $\mathbb{R}^3$ into unit cubes, and let $\mathcal{G}^* \subset \mathcal{G}$ be the grid cells intersected by at least one cube in $\mathcal{C}$; $|\mathcal{G}^*| \leq 8n$. Let $\mathcal{C}_G \subseteq \mathcal{C} := \{C \cap G \neq \varnothing \mid C \in \mathcal{C}\}$ for each grid cell $G \in \mathcal{G}^*$; $\sum_{G \in \mathcal{G}^*} |\mathcal{C}_G| \leq 8n$. For any grid cell $G \in \mathcal{G}^*$, any cube in $\mathcal{C}_G$ contains a vertex of G, which implies that $|\mathcal{C}_G|$ is bounded by a constant since the cubes of $\mathcal{C}$ have bounded depth. Therefore $\mathcal{U}(\mathcal{C}_G) \cap G = \mathcal{U} \cap G$ can be computed in $O(1)$ time. Then $\mathcal{U}$ can be computed by merging the portions within each grid cell of $\mathcal{G}^*$ in $O(n + \kappa) = O(n)$ time, where κ is the number of vertices on $\mathcal{U}$, which is bounded by $O(n)$ in this case. Computing $\mathcal{C}_G$ for all grid cells $G \in \mathcal{G}^*$ takes $O(n \log n)$ time, so the running time of the entire algorithm is $O(n \log n)$. Note that if the maximum distance between any two centers of cubes in $\mathcal{C}$ is polynomially bounded (i.e., is at most n^c for a constant $c > 0$) and $\lfloor x \rfloor$ can be computed in constant time for any $x \in \mathbb{R}$, computing the $\mathcal{C}_G$'s can be done in $O(n)$ time, which improves the overall running time to $O(n)$.

5 Conclusion

We have described algorithms to compute (the boundary of) the union of axis-aligned 3D cubes (or fat boxes) in general position in an output-sensitive manner. In particular, if the cubes have different sizes the union can be computed in $O(n \log^3 n + \kappa)$ time, where κ is the number of union vertices. If all cubes have the same size or they have bounded depth, then the union can be computed in $O(n \log^2 n)$ time, and if both conditions hold then the running time improves to $O(n \log n)$. We conclude by mentioning two open problems:

(i) Can the running time be improved to $O(n \log n + \kappa)$?

(ii) Can the union of a set of n cubes in $\mathbb{R}^d$ be computed in $O(n^{\lfloor d/2 \rfloor} + \kappa)$ time? In particular, can the union of n hypercubes in $\mathbb{R}^4$ be computed in $O(n \,\text{polylog}(n) + \kappa)$ time? Kaplan *et al.* [15] have shown that for a special case of orthants in $\mathbb{R}^d$, the union of n such orthants can be computed in $O(n + \kappa) \log^{d-1} n)$ time, but their algorithm does not extend to hypercubes.

References

1. P. K. Agarwal. An improved algorithm for computing the volume of the union of cubes. In *Proc. 26th Annu. Sympos. Comp. Geom.*, pages 230–239. ACM, 2010.
2. P. K. Agarwal, H. Kaplan, and M. Sharir. Computing the volume of the union of cubes. In *Proc. 23rd Annu. Sympos. Comp. Geom.*, pages 294–301. ACM, 2007.
3. P. K. Agarwal, H. Kaplan, and M. Sharir. Union of hypercubes and 3d minkowski sums with random sizes. *Discret. Comput. Geom.*, 65(4):1136–1165, 2021.
4. P. K. Agarwal, M. Sharir, and A. Steiger. Decomposing the complement of the union of cubes in three dimensions. In *Proc. 2021 ACM-SIAM Sympos. Discrete Algorithms*, pages 1425–1444. SIAM, 2021.
5. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *J. ACM*, 45(6):891–923, 1998.
6. J. Boissonnat, M. Sharir, B. Tagansky, and M. Yvinec. Voronoi diagrams in higher dimensions under certain polyhedral distance functions. *Discret. Comput. Geom.*, 19(4):485–519, 1998.
7. K. Bringmann. An improved algorithm for Klee's measure problem on fat boxes. *Comput. Geom.*, 45(5-6):225–233, 2012.
8. T. M. Chan. Klee's measure problem made easy. In *Proc. 54th Annu. IEEE Sympos. Found. Computer Science*, pages 410–419. IEEE Computer Society, 2013.
9. L. P. Chew, D. Dor, A. Efrat, and K. Kedem. Geometric pattern matching in d -dimensional space. *Discret. Comput. Geom.*, 21(2):257–274, 1999.
10. H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.
11. A. Efrat. Private communication.
12. R. H. Güting. An optimal contour algorithm for iso-oriented rectangles. *J. Algorithms*, 5(3):303–326, 1984.
13. S. Har-Peled. *Geometric Approximation Algorithms*. American Mathematical Society, 2011.
14. S. Har-Peled and K. Quanrud. Approximation algorithms for polynomial-expansion and low-density graphs. *SIAM J. Comput.*, 46(6):1712–1744, 2017.
15. H. Kaplan, N. Rubin, M. Sharir, and E. Verbin. Efficient colored orthogonal range counting. *SIAM J. Comput.*, 38(3):982–1011, 2008.
16. M. H. Overmars and C. Yap. New upper bounds in Klee's measure problem. *SIAM J. Comput.*, 20(6):1034–1045, 1991.
17. F. P. Preparata and M. I. Shamos. *Computational Geometry - An Introduction*. Springer, 1985.
18. D. Wood. The contour problem for rectilinear polygons. *Inf. Process. Lett.*, 19(5):229–236, 1984.
19. H. Yildiz and S. Suri. Computing Klee's measure of grounded boxes. *Algorithmica*, 71(2):307–329, 2015.