Computing the Volume of the Union of Cubes[*]

Pankaj K. Agarwal[†] Haim Kaplan[‡] Micha Sharir[§]

December 3, 2006

Abstract

Let $\mathcal{C}$ be a set of n axis-aligned cubes in $\mathbb{R}^3$, and let $\mathcal{U}(\mathcal{C})$ denote the union of $\mathcal{C}$. We present an algorithm that can compute the volume of $\mathcal{U}(\mathcal{C})$ in time $O(n^{4/3} \log n)$. The previously best known algorithm was by Overmars and Yap, which computes the volume of the union of boxes in $\mathbb{R}^3$ in $O(n^{3/2} \log n)$ time.

[*]Work by Pankaj Agarwal and Micha Sharir was supported by a grant from the U.S.-Israel Binational Science Foundation. Work by Pankaj Agarwal was also supported by by NSF under grants CCR-00-86013, EIA-98-70724, EIA-99-72879, EIA-01-31905, and CCR-02-04118. Work by Haim Kaplan was supported by Grant 975/06 from the Israel Science Fund. Work by Micha Sharir was also supported by NSF Grants CCR-00-98246 and CCF-05-14079, by Grant 155/05 from the Israel Science Fund, and by the Hermann Minkowski–MINERVA Center for Geometry at Tel Aviv University.

[†]Dept. Computer Science, Duke University, Durham, NC 27708-0129, USA. pankaj@cs.duke.edu

[‡]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel, haimk@post.tau.ac.il

[§]School of Computer Science, Tel Aviv University, Tel Aviv 69978, Israel, and Courant Inst. of Math. Sci., 251 Mercer Street, NYC, NY 10012, USA. michas@post.tau.ac.il

1 Introduction

Let $\mathcal{C}$ be a set of n axis-aligned cubes in $\mathbb{R}^3$, and let $\mathcal{U}(\mathcal{C})$ denote the union of $\mathcal{C}$. The problem studied in this paper is to compute the volume of $\mathcal{U}(\mathcal{C})$ efficiently. This is related to the well-known *Klee's measure problem*. In 1977 Victor Klee [11] had presented an $O(n \log n)$ time algorithm for computing the union of n intervals in $\mathbb{R}^1$ and had asked whether his algorithm was optimal. An $\Omega(n \log n)$ lower bound was proved by Fredmen and Weide [9]. Bentley [4] studied the two-dimensional version of Klee's measure problem. When extended to computing the volume of the union of n d-dimensional axis-aligned boxes, its running time is $O(n^{d-1} \log n)$. Later, van Leeuwen and Wood [10] improved the running time to $O(n^2)$ for $d = 3$. The problem lay dormant for a while until Overmars and Yap presented an algorithm with $O(n^{d/2} \log n)$ running time [12]. In spite of several attempts, no further progress was made on this problem, except for a somewhat simpler solution but with the same running time for $d = 3, 4$ [7], and for a more space-efficient algorithm [6]. See [1] for a brief history of the problem.

In contrast, there has been tremendous progress in the last decade on obtaining sharp bounds on the combinatorial complexity of the union (i.e., the number of faces of all dimensions on the boundary of the union) of objects. For example, Boissonnat *et al.* [5] proved that the combinatorial complexity of n axis-aligned cubes in $\mathbb{R}^d$ is $\Theta(n^{\lceil d/2 \rceil})$, and it is $\Theta(n^{\lfloor d/2 \rfloor})$ if all the cubes have the same size. Note that this bound is considerably better than the $\Theta(n^d)$ worst-case bound on the union of n axis-aligned boxes in $\mathbb{R}^d$. This suggests that it might be easier to compute the volume of the union of n cubes. Indeed, the volume of the union of n unit cubes in $\mathbb{R}^3$ can be computed in $O(n \log n)$ time, by computing their union explicitly (which has linear complexity). However this will not lead to an efficient algorithm for cubes of different sizes. Edelsbrunner [8] gave an inclusion-exclusion formula for computing the volume of the union of n balls (see also [2]). It might be possible to extend his approach to computing the volume of the union of cubes in $\mathbb{R}^3$, but the running time will be $\Omega(n^2)$ in the worst case.

In this paper we show that one can indeed exploit the special structure of cubes in $\mathbb{R}^3$ in order to compute the volume of their union more efficiently, and present the following result.

Theorem 1.1 *Let $\mathcal{C}$ be a set of n axis-aligned cubes in $\mathbb{R}^3$. The volume of $\mathcal{U}(\mathcal{C})$ can be computed in time $O(n^{4/3} \log n)$.*

The high-level approach of our algorithm is similar to that of [12], in the sense that it is also based on sweeping the space with a horizontal plane. The details are, however, more intricate, and exploit, sometimes in subtle ways, the fact that we are dealing with cubes, rather than boxes. We believe that our algorithm is far from being optimal, and that the running time can be improved to $O(n \operatorname{polylog}(n))$, but so far we have not been able to overcome all of the technical difficulties (which, as the reader might appreciate, are quite numerous).

The algorithm asserted in Theorem 1.1 is presented in the three following sections. Section 2 gives the high-level description of the algorithm, Section 3 goes into the more technical low-level details, and Section 4 presents further details of the data structure that maintains the union during the sweep.

2 The Global Structure

We assume that the given cubes are in *general position*. In particular, we assume that no plane support facets of two distinct cubes in $\mathcal{C}$. Let $z_1 < \cdots < z_{2n}$ be the (distinct) z-coordinates of the vertices of cubes in $\mathcal{C}$, sorted in increasing order. We sweep a horizontal plane Π in the $(+z)$-direction from $-\infty$ to $+\infty$, stopping at each z_i. Let $\Pi(t)$ denote the horizontal plane at $z = t$. For each $1 \le i < 2n$, the cross-section

$\mathcal{U}(\mathcal{C}) \cap \Pi(z)$ is the same for all $z \in (z_i, z_{i+1})$. Let a_i denote the area of this cross-section. Then

$$\mathrm{Vol}\, \mathcal{U}(\mathcal{C}) = \sum_{i=1}^{2n-1} a_i(z_{i+1} - z_i).$$

We thus need to maintain a_i as we sweep the horizontal plane. The intersection of $\Pi(z)$ with $\mathcal{U}(\mathcal{C})$ is the union of a set $\mathcal{S}$ of squares that changes dynamically—a square is added to the intersection when Π sweeps through the bottom facet of its corresponding cube, and is removed from the intersection when Π sweeps through the top facet of its cube. We describe a data structure that maintains, in $O(n^{1/3} \log n)$ amortized time, the area of the union of $\mathcal{S}$, denoted by $\mathrm{Area}\, \mathcal{U}(\mathcal{S})$, as we insert a square into $\mathcal{S}$ or delete a square from $\mathcal{S}$ during the sweep. This implies Theorem 1.1. Our procedure exploits, in a crucial though subtle way, the special (obvious) property that the *life-time* of a square of side length h (regarding the z-direction as "time") is also h time units.

In our case, we know in advance the set $\mathcal{V} \subset \mathbb{R}^2$ of vertices of all the squares that will ever be inserted into $\mathcal{S}$. That is, $\mathcal{V}$ is the set of the xy-projections of the vertices of the given cubes, and we have $|\mathcal{V}| = 4n$. Let $\mathcal{B}$ be the smallest axis-parallel rectangle containing $\mathcal{V}$. We choose the parameter $s = n^{1/3}$, and partition $\mathcal{B}$ into s rectangles $B_1, \ldots, B_s$, called *slabs*, by vertical lines (parallel to the y-axis), so that the interior of each B_i contains at most $4n/s = O(n^{2/3})$ vertices of $\mathcal{V}$; see Figure 1. Next, we partition each B_i into s rectangles, called *cells*, by lines parallel to the x-axis, so that the interior of each cell contains at most $4n/s^2 = O(n^{1/3})$ vertices. For $1 \le i \le s$, we maintain $\alpha_i = \mathrm{Area}\, \mathcal{U}(\mathcal{S}) \cap B_i$, using a binary tree $\mathcal{T}_i$ with s leaves. Each node v of $\mathcal{T}_i$ is associated with a rectangle $\square_v$ contained in B_i and touching its two vertical sides. For the ith leftmost leaf v of $\mathcal{T}_i$, $\square_v$ is the ith bottom-most cell of B_i. For an interior node v with children w and z, $\square_v = \square_w \cup \square_z$. For a node $v \in \mathcal{T}_i$, let $\mathcal{S}_v \subseteq \mathcal{S}$ be the set of squares whose boundaries intersect the interior of $\square_v$, and $\mathcal{S}_v^* \subseteq \mathcal{S}$ be the set of squares that contain $\square_v$ but not $\square_{p(v)}$ (where $p(v)$ is the parent of v). At each leaf v of $\mathcal{T}_i$ we store the respective set $\mathcal{S}_v$, and we also store $\sigma_v = |\mathcal{S}_v^*|$ at each node v. For a node $v \in \mathcal{T}_i$, let $\alpha_v = \mathrm{Area}\, \mathcal{U}(\mathcal{S}_v \cup \mathcal{S}_v^*) \cap \square_v$. If v is a leaf, we compute and update α_v using the algorithm described in Section 3. For an interior node v with children w and z, we have

$$\alpha_v = \begin{cases} \mathrm{Area}\, \square_v & \text{if } \sigma_v \ge 1, \\ \alpha_w + \alpha_z & \text{if } \sigma_v = 0. \end{cases} \tag{1}$$

When we insert a square S, we first find all the slabs B_i that S meets. Then, for each of these B_i, we find the leaves v of $\mathcal{T}_i$ such that $\square_v \cap \partial S \neq \emptyset$. For each such v, we insert S into $\mathcal{S}_v$ and update α_v using the algorithm described in Section 3. If B_i does not contain a vertex of S, then S is inserted into at most two leaves w and z, such that $\square_w$ and $\square_z$ intersect the horizontal edges of S. Next, we find all $O(\log n)$ nodes u in $\mathcal{T}_i$ which lie between w and z, and whose parents lie along the two paths of $\mathcal{T}_i$ to w and to z, so that $\square_u \subseteq S$ but $\square_{p(u)} \not\subseteq S$. For each such u, we increment the value of σ_u and set $\alpha_u = \mathrm{Area}\, \square_u$. If B_i contains a vertex of S, then S may have to be inserted into many (perhaps all) leaves of $\mathcal{T}_i$, and $S \notin \mathcal{S}_v^*$ for any $v \in \mathcal{T}_i$. For each leaf v for which $\square_v \cap \partial S \neq \emptyset$, we update α_v, using the algorithm of Section 3.

Finally, using (1) in a bottom-up manner, we update the values α_u for all ancestors u of any node v that has been updated. We repeat this procedure for each of the slabs B_i, and return the value of $\sum_{i=1}^{s} \alpha_{\mathrm{root}(\mathcal{T}_i)}$. A square is deleted from $\mathcal{S}$ in a similar manner.

Let v be a leaf of $\mathcal{T}$ such that $S \in \mathcal{S}_v$. We show in the next section (cf. Lemma 3.1) that if $\square_v \cap \partial S \neq \emptyset$, then (i) α_v can be updated in $O(\log n)$ amortized time if $\square_v$ does not contain a vertex of S in its interior, and (ii) α_v can be updated in $O((n/s^2) \log n) = O(n^{1/3} \log n)$ amortized time if $\square_v$ does contain a vertex of S in its interior. There are at most four cells that contain a vertex of S, and there are at most $4s$ cells that

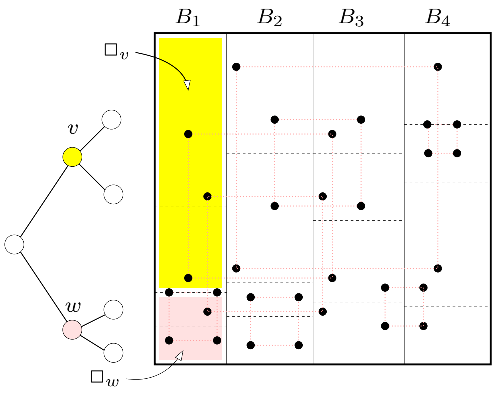

Figure 1: Partition of B into cells and the tree $\mathcal{T}_1$.

intersect ∂S, so we spend a total of $O((s + n/s^2)\log n) = O(n^{1/3}\log n)$ amortized time in updating the areas at the leaves of the trees $\mathcal{T}_i$. We then update the ancestors of the updated leaves. In the worst case, we visit all the nodes of at most two $\mathcal{T}_i$'s—if B_i contains a vertex of S—and $O(\log n)$ nodes of any other $\mathcal{T}_i$. We spend $O(1)$ time at each of these ancestor nodes. Hence, the total amortized time spent in updating $\mathrm{Area}\,\mathcal{U}(\mathcal{S})$ when a square is inserted or deleted is also $O(n^{1/3}\log n)$. That is, $\mathrm{Area}\,\mathcal{U}(\mathcal{S})$ can be updated in $O(n^{1/3}\log n)$ amortized time after each update operation, and Theorem 1.1 follows.

Remark. We believe that the running time of the algorithm can be improved to $O(n\,\mathrm{polylog}(n))$ by using a recursive binary partioning of $\mathcal{B}$ and maintaining a similar (but more involved) information at each cell in the recursive partition. However we have not succeeded in overcoming all the technical difficulties in updating the information at each cell in (even amortized) $O(\log n)$ time when a square is inserted or deleted.

3 Mantaining the Union within a Cell

Let $\square = [x_0, x_1] \times [y_0, y_1]$ be a fixed cell in the partition of $\mathcal{B}$, which, as we recall, is an axis-parallel rectangle in $\mathbb{R}^2$, where we assume, without loss of generality, that $x_1 - x_0 \geq y_1 - y_0$ (handling cells with $x_1 - x_0 < y_1 - y_0$ is done in a fully symmetric manner, switching the roles of the x- and y-axes). Let $p_0 = (x_0, y_0), p_1 = (x_1, y_0), p_2 = (x_1, y_1)$, and $p_3 = (x_0, y_1)$ be its vertices in counterclockwise order. Let $\mathcal{S}$ be a set of squares whose boundaries intersect $\square$. We describe a data structure for maintaining the area $\alpha_\square$ of $\mathcal{U}_\square = \mathcal{U}(\mathcal{S}) \cap \square$ under insert/delete operations on $\mathcal{S}$, where we also assume that the life-span of each square in $\mathcal{S}$ is equal to its side length. Recall that the set $\mathcal{S}$ is updated when the sweep plane passes through a top or a bottom facet of a cube in $\mathcal{C}$.

Since the boundary of each square in $\mathcal{S}$ intersects $\square$, and the x-span of $\square$ is at least as large as its y-span, no square $S \in \mathcal{S}$ can intersect both left and right edges of $\square$ without fully containing either the top or bottom edge of $\square$. We partition $\mathcal{S}$ into the following subsets (see Figure 2):

Upper rim. The set of squares, denoted by $\mathbb{U}$, that contain the top edge of $\square$. We store $\mathbb{U}$ in a list sorted in decreasing order of the y-coordinates of their bottom edges.

Lower rim. The set of squares, denoted by $\mathbb{L}$, that contain the bottom edge of $\square$. We store $\mathbb{L}$ in a list sorted in increasing order of the y-coordinates of their top edges.

Pillars. The set of squares, denoted by $\mathbb{P}$, that intersect both top and bottom edges of $\square$.

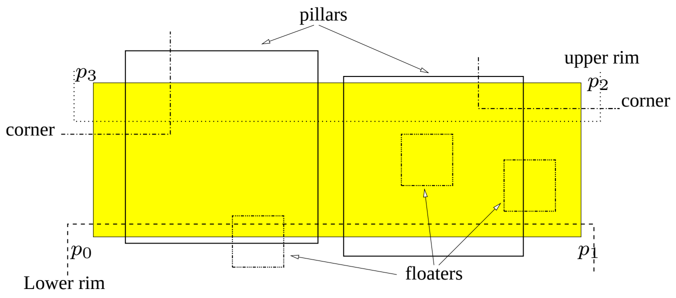

Figure 2: Partition of $\mathbb{S}$ into various categories.

Corners. The set of squares, denoted by $\mathbb{C}$, that contain exactly one vertex of $\square$; exactly one vertex of each corner square lies in $\square$.

Floaters. The set of remaining squares, denoted by $\mathbb{F}$; at least two (i.e., either two or four) of the vertices of each floater square lie in $\square$.

The first three types of squares are called *long*, and the last two types are called *short*.[1] The *floor* of $\square$ is the top edge of the last square in the lower rim (i.e., the highest edge in the lower rim), and the *ceiling* is the bottom edge of the last square (the lowest edge) in the upper rim.

Informally, inserting/deleting a short square is "easier", since their number, over all cells, is only $O(n)$, so we can afford $O(n^{1/3})$ (amortized) time to process a short square. For example, we can afford to (and indeed we will) recompute the union of the corners or of the floaters when we insert or delete one of these squares. In contrast, inserting/deleting a long square (rim or pillar) is more challenging, since we want to do it in only $O(\log n)$ (amortized) time.

The main technical complication in our solution is that, while it is fairly easy to maintain the area of the union of each class of squares separately, it is much more involved to maintain the area of the combined union. Our approach is to maintain this latter area as the sum of areas of disjoint portions of $\square$—the area covered by the rims, the area covered by the pillars but not by the rims, the area covered by the corner squares but not by the pillars or rims, and finally the area covered by the floaters but not by any other square. maintaining these disjoint areas is somewhat tricky; the high-level details are given in this section, and the low-level details in the following section.

We call a (rectilinear) polygon *staircase* if it consists of a rectilinear chain that is both x- and y-monotone (in each coordinate it can be either increasing or decreasing), and the endpoints of the chain are connected together by a horizontal and a vertical edge; see Figure 3 (i). The common endpoint of the horizontal and the vertical edge is called the *apex* of the polygon. Since the complexity of the union of a set of axis-parallel squares is linear, $\mathcal{U}(\mathbb{C}) \cap \square$ has $O(|\mathbb{C}|)$ vertices. We can decompose $\mathcal{U}(\mathbb{C}) \cap \square$ into four pairwise-disjoint staircase polygons, P_0, P_1, P_2, P_3, with a total of $O(|\mathbb{C}|)$ vertices, such that the apex of P_i is the corner p_i of $\square$ (see Figure 3 (ii)). (Informally, P_i is composed of the corner squares that contain p_i, but since other squares can "nibble off" some portions of these squares, as is illustrated in the figure, P_i may be smaller than the union of its squares. Also, the P_i's are not uniquely defined, but, since they will be recomputed from scratch when we insert or delete a square into $\mathbb{C}$, it does not matter.) We decompose each P_i into a

[1]This is somewhat of a misnomer for corner squares, which can be quite large compared with $\square$, but we still think of them as short since they have a vertex inside $\square$.

set $\tilde{\mathbb{C}}_i$ of rectangles by computing the vertical decomposition of P_i (i.e., drawing a vertical edge from each reflex vertex of P_i within $\square$ until it touches a horizontal edge of $\square$; see Figure 3 (iii)). Set $\tilde{\mathbb{C}} = \bigcup_{i=0}^{3} \tilde{\mathbb{C}}_i$. By construction, $\mathcal{U}(\tilde{\mathbb{C}}) = \mathcal{U}(\mathbb{C}) \cap \square$.

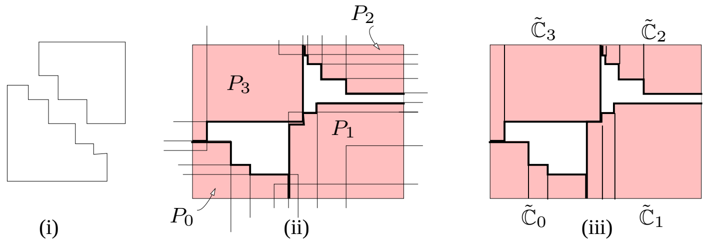

Figure 3: (i) Staircase polygons. (ii) Decomposition of $\mathcal{U}(\mathbb{C})$ into four staircase polygons $P_0, \ldots, P_3$. (iii) The decomposition $\tilde{\mathbb{C}}_i$ of each P_i into rectangles.

Next, we compute $(\mathcal{U}(\mathbb{F}) \cap \square) \setminus \mathcal{U}(\mathbb{C})$, i.e., the portion of $\mathcal{U}(\mathbb{F}) \cap \square$ that lies outside $\mathcal{U}(\mathbb{C})$, and partition it into pairwise-disjoint rectangles by computing its vertical decomposition. Let $\mathcal{R} = \{R_1, \ldots, R_u\}$ be the set of the rectangles in the resulting decomposition. Since $\mathcal{U}(\mathbb{F})$ and $\mathcal{U}(\mathbb{C})$ have $O(|\mathbb{F}|)$ and $O(|\mathbb{C}|)$ vertices, respectively, $|\mathcal{R}| = O(|\mathbb{C}| + |\mathbb{F}|)$. See Figure 4. We call a rectangle of $\mathcal{R}$ *stalactite* (resp., *stalagmite*) if it intersects the ceiling (resp., floor) of $\square$; a rectangle may be both a stalactite and a stalagmite. Let $\mathbb{S}^-$ (resp., $\mathbb{S}^+$) denote the set of stalagmites (resp., stalactites) in $\mathcal{R}$.

We store the horizontal edges of $\mathbb{F}$ (or, more precisely, the rectangles in $\mathcal{R}$) in a list Λ, sorted by their y-coordinates. We also maintain the following auxiliary data:

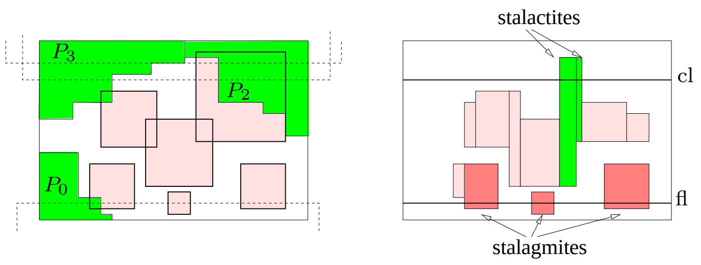

Figure 4: (i) Squares in $\mathbb{F}$; the darkly-shaded region is $\mathcal{U}(\mathbb{C}) \cap \square$, and the lightly-shaded region is $(\mathcal{U}(\mathbb{F}) \cap \square) \setminus \mathcal{U}(\mathbb{C})$; the rim squares are drawn as dashed. (ii) Rectangles in $\mathcal{R}$; the dark shaded rectangles are the stalagmites and stalactites.

π: The length of the portion of the top edge of $\square$ (or any other horizontal line intersecting $\square$) covered by the pillars.

φ: The area of $\mathcal{U}(\mathcal{R})$ not covered by the long squares (i.e., pillars, upper rim, and lower rim).

fl: the y-coordinate of the floor of $\square$.

cl: the y-coordinate of the ceiling of $\square$.

λ_c: The length of the ceiling covered by the stalactites but not by the pillars.

λ_f: The length of the floor covered by the stalagmites but not by the pillars.

By definition, λ_f (resp., λ_c) measures exactly the portion of the floor (resp., ceiling) covered by the floaters but by no other square.

For a y-interval Δ, let W_Δ denote the rectangle $[x_0, x_1] \times \Delta$, and let $\xi(\Delta) = \text{Area}[\mathcal{U}(\tilde{\mathbb{C}}) \setminus \mathcal{U}(\mathbb{P})] \cap W_\Delta$. We construct a segment tree $\mathbb{T}$ on the x-projections of the rectangles (or squares) in $\mathbb{P} \cup \mathcal{R} \cup \tilde{\mathbb{C}}$, so as to maintain the above data and to answer queries of the form: Given a y-interval $\Delta \subseteq [y_0, y_1]$, return $\xi(\Delta)$. The values λ_c, λ_f, φ, π, will get updated each time we update $\mathbb{T}$.

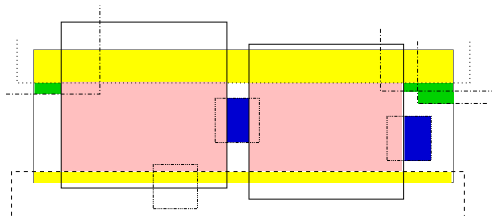

Figure 5: Each term of (2) is shown in a different shade.

Assuming that we can maintain the above data and query processing as the squares of $\mathcal{S}$ are inserted and deleted, the area $\alpha_\square$ (which, as we recall, is the area of the portion of $\square$ covered by the squares of $\mathcal{S}$, whose boundaries cross $\square$) can be computed in $O(\log n)$ time as follows. If $\text{cl} \le \text{fl}$, then $\alpha_\square = \text{Area}\,\square$. Otherwise, we compute $\xi(\Delta)$, for $\Delta = [\text{fl}, \text{cl}]$, by performing a query on $\mathbb{T}$. Then, as is easily verified,

$$\alpha_\square = (x_1 - x_0)[(\text{fl} - y_0) + (y_1 - \text{cl})] + (\text{cl} - \text{fl})\pi + \xi(\Delta) + \varphi. \tag{2}$$

Indeed, the first term is the area of the region covered by the upper rim and lower rim squares. The second term is the area of the region covered by the pillars but not by any rim square. The third term is the area of the region covered by the corner squares but not by any rim square or pillar, and the fourth term is the area of the region covered by the floater squares but not by any other type of squares (recall that, by definition, $\mathcal{R}$ is disjoint from the corner squares). See Figure Figure 5.

Inserting/Deleting a short square. Suppose we want to insert a short square S into $\mathcal{S}$ or delete S from $\mathcal{S}$. Let $\mu \le 4n/s^2 = O(n^{1/3})$ denote the maximum number of short squares ever present in $\mathcal{S}$. If $S \in \mathbb{C}$, we recompute $\tilde{\mathbb{C}}$ in $O(\mu \log n)$ time. We delete the old rectangles of $\tilde{\mathbb{C}}$ from $\mathbb{T}$ and insert the new ones, where each update takes $O(\log n)$ time (this will be shown in Lemma 4.1 below). Next, in both cases where S is a floater or a corner, we re-compute $\mathcal{R}$ in $O(\mu \log n)$ time, and reconstruct the list Λ. We delete all old rectangles of $\mathcal{R}$ from $\mathbb{T}$ and insert each new rectangle of $\mathcal{R}$ into $\mathbb{T}$, in a total of $O(\mu \log n)$ time (again, Lemma 4.1 below will show this). The data stored at the root of $\mathbb{T}$ provides the new values of π, φ, λ_c, and λ_f. Deleting a short square can be done in the same manner. Finally, using (2), we compute the new value of $\alpha_\square$ in additional $O(\log n)$ time (dominated by the cost of the appropriate query). The total time spent in inserting or deleting a short square is thus at most $c\mu \log n$, for some constant $c > 0$.

Inserting/Deleting a long square. Suppose we want to insert a long square S. If S is a pillar, we simply insert it into $\mathbb{T}$ in $O(\log n)$ time (cf. Lemma 4.1). If S is a lower-rim square, we first insert S into the sorted sequence $\mathbb{L}$. If S is not the topmost square, it does not affect $\mathcal{U}_\square$, and we stop. Otherwise, let χ be the y-coordinate of the top edge of S. We raise the floor continuously from fl to χ, stopping at each horizontal edge of a square in $\mathbb{F}$ (or, more precisely, of a rectangle in $\mathcal{R}$) that the sweep encounters, and updating φ, λ_f, and fl, until we reach χ. In more detail, we find the first edge in Λ that lies above the current floor of $\Box$, and then scan Λ from this edge upwards, processing each edge that lies below χ. Consider such an edge e at y-coordinate τ. Since the floor has not crossed any edge of Λ since the last event, the value of φ has decreased by $\lambda_f(\tau - \text{fl})$ between the preceding and current events. Therefore, we set $\varphi := \varphi - \lambda_f(\tau - \text{fl})$ and $\text{fl} := \tau$. This update is applied also at the end $\tau = \chi$ of the sweep. We next need to update (for $\tau < \chi$) the value of λ_f, because we now have one additional or one fewer stalagmite. For this, we simply invoke the procedure described above to insert/delete a short square (even though no square is actually inserted or deleted at the moment). This seemingly expensive step will be justified in the amortized analysis given below. Finally, we compute the new value of $\alpha_\Box$ in $O(\log n)$ time, using (2).

If we delete the highest square of the lower rim, we need to lower the floor. We follow the same algorithm except that at each event, we set $\varphi := \varphi + \lambda_f(\text{fl} - \tau)$. We can lower and raise the ceiling in a similar manner. If the algorithm sweeps across κ horizontal edges of $\mathcal{R}$, these updates take at most $c(\kappa\mu + 1)\log n$ time.

Amortized analysis. We now analyze the amortized running time of each update operation using a credit-debit method. This is the only step[2] where we use the fact that the squares are being inserted or deleted while sweeping a plane through a set of cubes in $\mathbb{R}^3$. We assign $8c\mu\log n$ credits to each square in $\mathbb{F}$ when it is inserted, and use them to pay for the cost of inserting and deleting upper/lower-rim squares. Since the update time of a short square is $\leq c\mu\log n$, the amortized update time is also $\leq c\mu\log n$. The amortized update time of a pillar is $O(\log n)$ (recall that these running times will be established in Lemma 4.1 below). The actual update time of a lower or upper rim square S is at most $c(\kappa\mu + 1)\log n$, where κ is the number of edges in Λ crossed by the rising/descending-floor (or descending/rising-ceiling) sweep-line algorithm. We charge $c\mu\log n$ units to each square in $\mathbb{F}$ whenever the sweep line passes through one of its edges, so the amortized (i.e., uncharged portion of the) update time of a square in the upper/lower rim is also $O(\log n)$.

We now have to prove that each square in $\mathbb{F}$ always has sufficient amount of credits to pay for updating the upper/lower rim. The key observation is that the size of a lower or upper rim square is bigger than that of a floater. Recall that a square is inserted when the sweep plane reaches the bottom facet of its cube C and is deleted when it reaches the top facet of C. Therefore if a lower/upper rim square S is inserted *after* a floater S', then S' is deleted *before* S is deleted. This implies that if the floor was raised above an edge e of S', during the insertion of S, it is lowered below that height (while deleting S or some other square after S has been deleted) only after S' has already been deleted. Hence, the floor sweeps across e, while e is alive, at most twice: it may be lowered below e once (during the deletion of a lower-rim square that was inserted *before* S' was inserted), and then it may be raised above e once. The same is true for the upper rim squares. Since S' has two horizontal edges, S' is charged at most eight times by the update procedure for the upper/lower rim, thereby implying that $8c\mu\log n$ credits are sufficient for each square in $\mathbb{F}$ to pay for inserting/deleting a square in the upper or lower rim. We thus conclude the following.

Lemma 3.1 *The amortized update time of a long square is $O(\log n)$, and the amortized (and worst case) update time of a short square is $O(\mu\log n)$, where $\mu \leq 4n/s^2 = O(n^{1/3})$ is the maximum number of short squares in $\Box$ at any time.*

4 Maintaining the Segment Tree

To complete the analysis, we now describe the segment tree $\mathbb{T}$ constructed on the x-projections of rectangles in $\mathbb{P} \cup \tilde{\mathbb{C}} \cup \mathcal{R}$. We assume, as is indeed the case in our scenario, that we know the endpoints of the x-

[2]Of course, we are also strongly using the fact that the horizontal cross-sections of the cubes are squares.

projections of all squares in $\mathcal{S}$ in advance, so that the primary tree structure of $\mathbb{T}$ is constructed over all these segments, and remains fixed. Each node v of $\mathbb{T}$ is associated with an x-interval δ_v and a corresponding rectangle $\Box_v = \delta_v \times [y_0, y_1]$. If w, z are the two children of v, then $\delta_v = \delta_w \cup \delta_z$. For a node $v \in \mathbb{T}$, let $\mathcal{R}_v \subseteq \mathcal{R}$ (resp., $\mathbb{P}_v \subseteq \mathbb{P}$, $\tilde{\mathbb{C}}_{iv} \subseteq \tilde{\mathbb{C}}_i$, for $0 \le i \le 3$) be the set of rectangles of the respective classes whose x-projections contain δ_v but not $\delta_{p(v)}$. Set $\mathcal{R}^*_v = \bigcup_w \mathcal{R}_w$, $\mathbb{P}^*_v = \bigcup_w \mathbb{P}_w$, and $\tilde{\mathbb{C}}^*_{iv} = \bigcup_w \tilde{\mathbb{C}}_{iw}$, over all descendents w of v (including v itself). We store the set $\mathcal{R}_v$ and the size $|\mathbb{P}_v|$ at each node v. We also maintain the following auxiliary information at each node v of $\mathbb{T}$.

$\pi(v)$: The length of the portion of δ_v covered by the x-projections of the pillars in $\mathbb{P}^*_v$.

$\varphi(v)$: Area of $\mathcal{U}(\mathcal{R}^*_v) \cap \Box_v$ not covered by the long squares (pillars, upper rim, lower rim) of $\mathcal{S}$.

$h(v)$: Length of the left edge of $\Box_v$ covered by the rectangles in $\mathcal{R}_v$ but not by the squares in the upper or lower rim.

$\lambda_f(v)$: Length of the floor covered by the x-projections of the stalagmites in $\mathcal{R}^*_v$ but not by the pillars.

$\lambda_c(v)$: Length of the ceiling covered by the x-projections of the stalactites in $\mathcal{R}^*_v$ but not by the pillars.

$\bar{\xi}_i(v)$: $\mathrm{Area}([\mathcal{U}(\tilde{\mathbb{C}}^*_{iv}) \setminus \mathcal{U}(\mathbb{P}^*_v)] \cap \Box_v)$, for $i = 0, \dots, 3$.

$J_i(v)$: The vertical segment which is the y-projection of the portion of the rectilinear chain of the staircase polygon P_i that lies inside $\Box_v$. If $\Box_v$ intersects only one horizontal edge of the chain, then $J_i(v)$ is a singleton. The values of $J_i(v)$ along the leaves are ordered because $P_i(v)$ is a staircase, and therefore monotone, polygon.

$\mathrm{fl}(v)$: The most recently recorded value of fl at v.

$\mathrm{cl}(v)$: The most recently recorded value of cl at v.

We remark right away that not all these values are correctly maintained at all times at v, but they are maintained in such a way that it is easy to reset them (in $O(1)$ time) to the correct values upon demand—see below.

The values $\pi(r)$, $\varphi(r)$, $\lambda_f(r)$, and $\lambda_c(r)$, stored at the root r of $\mathbb{T}$ give the values of $\pi, \varphi, \lambda_f, \lambda_c$ that we maintain for $\Box$, as required by the algorithm described in Section 3. Immediately after an update of $\mathbb{T}$, these values at r are correct, although the value of φ can deviate from the value of $\varphi(r)$ after inserting ot deleting a lower-rim or an upper-rim square. However, the value can be restored to its correct value whenever $\mathbb{T}$ is accessed, as explained below.

Let ρ_v be the rectangle $\delta_v \times [\mathrm{fl}, \mathrm{cl}]$, assuming $\mathrm{fl} \le \mathrm{cl}$. For a rectangle ρ, let $\mathrm{Ht}(\rho)$ denote its height. Since, by construction, the rectangles in $\mathcal{R}_v$ are pairwise disjoint, and each of them extends from the left edge to the right edge of $\Box_v$, we have

$$h(v) = \sum_{R \in \mathcal{R}_v} \mathrm{Ht}(R \cap \rho_v). \qquad (3)$$

Note that $h(v)$ depends only on $\mathcal{R}_v$ and not on the other rectangles in the sets $\mathcal{R}_w$, for *proper* descendants w of v, nor does it depend on any other type of squares. Moreover, there is at most one rectangle in R_v that intersects the floor (resp., ceiling). Hence, $\mathrm{Ht}(R \cap \rho_v)$ is either 0 or $\mathrm{Ht}(R)$ except for at most two rectangles in $\mathcal{R}_v$.

Put $\ell_v := |\delta_v|$, for a node v of $\mathbb{T}$. Let w and z be the children of an interior node v. The following equalities are obvious (see Figure 6):

$$\pi(v) = \begin{cases} \ell_v & \text{if } \mathbb{P}_v \neq \emptyset, \\ \pi(w) + \pi(z) & \text{otherwise.} \end{cases} \tag{4}$$

$$\varphi(v) = \begin{cases} 0 & \text{if } \mathbb{P}_v \neq \emptyset, \\ \varphi(w) + \varphi(z) + (\ell_v - \pi(v))h(v) & \text{otherwise.} \end{cases} \tag{5}$$

$$\lambda_f(v) = \begin{cases} 0 & \text{if } \mathbb{P}_v \neq \emptyset, \\ \ell_v & \text{if } \mathbb{P}_v = \emptyset, \mathcal{R}_v \cap \mathbb{S}^- \neq \emptyset, \\ \lambda_f(w) + \lambda_f(z) & \text{otherwise.} \end{cases} \tag{6}$$

$$\lambda_c(v) = \begin{cases} 0 & \text{if } \mathbb{P}_v \neq \emptyset, \\ \ell_v & \text{if } \mathbb{P}_v = \emptyset, \mathcal{R}_v \cap \mathbb{S}^+ \neq \emptyset, \\ \lambda_c(w) + \lambda_c(z) & \text{otherwise.} \end{cases} \tag{7}$$

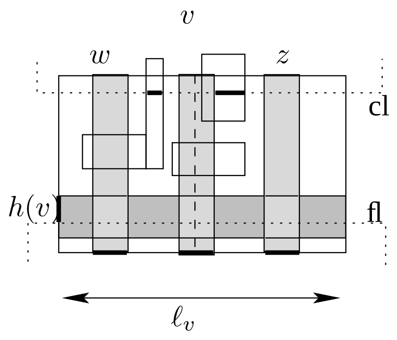

Figure 6: Illustrating (4)-(7) at a node v. $\pi(v)$ is the highlighted portion of the bottom edge; $h(v)$ is highlighted on the left edge. The small rectangles belong to $\mathcal{R}_w$ for proper descendants w of v and are disjoint from the rectangle of $\mathcal{R}_v$ that determine $h(v)$. $\lambda_f(v) = \ell_v$ and $\lambda_c(v)$ is highlighted on the ceiling.

If $\tilde{\mathbb{C}}_{iv} \neq \emptyset$, then it consists of a single rectangle, say C_{iv}, and let χ_{iv} denote the y-coordinate of its horizontal edge that lies in the interior of $\square_v$. Since the rectangles in $\tilde{\mathbb{C}}_i$ are pairwise disjoint and a vertical line intersects at most one rectangle of $\tilde{\mathbb{C}}_i$, $\tilde{\mathbb{C}}_{iv} \neq \emptyset$ implies that $\tilde{\mathbb{C}}^*_{iv} = \tilde{\mathbb{C}}_{iv} = \{C_{iv}\}$. Hence,

$$\bar{\xi}_i(v) = \begin{cases} 0 & \text{if } \mathbb{P}_v \neq \emptyset, \\ (\ell_v - \pi(v))\,\mathrm{Ht}(C_{iv}) & \text{if } \mathbb{P}_v = \emptyset \text{ and } \tilde{\mathbb{C}}_{iv} \neq \emptyset, \\ \bar{\xi}_i(w) + \bar{\xi}_i(z) & \text{otherwise.} \end{cases} \tag{8}$$

$$J_i(v) = \begin{cases} [\chi_{iv}, \chi_{iv}] & \text{if } \tilde{\mathbb{C}}_{iv} \neq \emptyset, \\ \mathrm{conv}(J_i(w) \cup J_i(z)) & \text{otherwise.} \end{cases} \tag{9}$$

We can use simpler variants of the above equations, which do not involve the recursive terms, for the values of these quantities at the leaf nodes; for example, $\varphi(v)$ is 0 if $\mathbb{P}_v \neq \emptyset$, and is $\ell_v h(v)$ otherwise.

Whenever a node v is updated, we call a subroutine UPDATEAUX(v) that implements (4)–(9), to recompute $\pi(v), \varphi(v), \lambda_f(v), \lambda_c(v), \bar{\xi}_i(v)$, and $J_i(v)$.

The data structure always maintains the correct values of $\pi(v), \lambda_c(v), \lambda_f(v), \bar{\xi}_i(v)$, and $J_i(v)$ at all nodes, but the values of $\mathrm{fl}(v), \mathrm{cl}(v), \varphi(v), h(v)$ may be incorrect because the updating of the floor and ceiling of $\square$ does not always reach all the nodes of $\mathbb{T}$ in "real time". For example, if the floor is raised when we insert a square of the lower rim, so that no edge in Λ lies between the new and old floor, then we do not udpate $\mathbb{T}$ at all, and thus the stored values of $\mathrm{fl}(v), \mathrm{cl}(v), \varphi(v)$, and $h(v)$ are not modified, even though the real values have changed. Even when $\mathbb{T}$ gets updated, when the floor or ceiling sweeps through a horizontal floater edge, not all the nodes of $\mathbb{T}$ "get the news"—it is too expensive to broadcast the changes explicitly to all nodes of $\mathbb{T}$. Instead, we update them in a lazy manner, so that the following two invariants are maintained:

(I1) *For any node $v \in \mathbb{T}$, none of the y-coordinates of the horizontal edges of rectangles in $\mathcal{R}_v^*$ lie between fl and $\mathrm{fl}(v)$ or between cl and $\mathrm{cl}(v)$.*

(I2) The value of $\varphi(v)$ *gives the area of $\mathcal{U}(\mathcal{R}_v^*) \cap \square_v$ not covered by the long squares of $\mathcal{S}$, under the assumption that the floor (resp., ceiling) is at $\mathrm{fl}(v)$ (resp., $\mathrm{cl}(v)$).*

To ensure that these quantities at a node v become correct whenever we access v, we apply the following two straightforward subroutines, before manipulating any other data at v.

```
ADJUSTFLOOR($v$)
    $\varphi(v) = \varphi(v) - \lambda_f(v)[\mathrm{fl} - \mathrm{fl}(v)]$
    if $\mathcal{R}(v) \cap \mathbb{S}^- \neq \emptyset$
        $h(v) = h(v) - [\mathrm{fl} - \mathrm{fl}(v)]$
    $\mathrm{fl}(v) = \mathrm{fl}$
```

```
ADJUSTCEILING($v$)
    $\varphi(v) = \varphi(v) - \lambda_c(v)[\mathrm{cl}(v) - \mathrm{cl}]$
    if $\mathcal{R}(v) \cap \mathbb{S}^+ \neq \emptyset$
        $h(v) = h(v) - [\mathrm{cl}(v) - \mathrm{cl}]$
    $\mathrm{cl}(v) = \mathrm{cl}$
```

By invariant (I1), there is no horizontal edge of $\mathcal{R}_v^*$ between fl and $\mathrm{fl}(v)$, so if we raise or lower the floor from $\mathrm{fl}(v)$ to fl then the value of $\varphi(v)$ decreases by the amount $\lambda_f(v)[\mathrm{fl} - \mathrm{fl}(v)]$. A similar argument justifies the updating applied to $h(v)$, and for the ceiling. This, in conjunction with (I2), implies that, after executing these procedures, $\mathrm{fl}(v)$, $\mathrm{cl}(v)$, $\varphi(v)$ and $h(v)$ have their correct values.

Generic update procedure. We insert or delete a rectangle of $\mathcal{S}$ using a slight variant of the standard update procedure for segment trees [3], so that the auxiliary information stored at each node is updated correctly and invariant (I1) is maintained. The following pseudo-code describes the generic procedure GENERICUPDATE to update $\mathbb{T}$ when a rectangle S is inserted into or deleted from $\mathcal{S}$. We use there $\mathrm{L}(v)$ (resp., $\mathrm{R}(v)$) to denote the left (resp., right) child of v.

```
GENERICUPDATE($v$, $S$)

  ADJUSTFLOOR($v$), ADJUSTCEILING($v$)
  $I := x$-projection of $S$
  if $\delta_v \subseteq I$
      Update information at $v$            ($\star$)
    else if $\delta_v \cap I \neq \emptyset$
      GENERICUPDATE($\mathrm{L}(v)$, $S$)
      GENERICUPDATE($\mathrm{R}(v)$, $S$)
      UPDATEAUX($v$)
    endif
```

The line ($\star$) depends on the type of rectangle that is being inserted or deleted. The specific actions taken at this line for each type of rectangle are described in the Appendix. The details of the query algorithm (finding $\xi(\Delta) = \mathrm{Area}[\mathcal{U}(\tilde{\mathbb{C}}) \setminus \mathcal{U}(\mathbb{P})] \cap W_\Delta$, for a y-interval Δ, and W_Δ the horizontal strip of Section 3 that corresponds to Δ.) are also deffered to the Appendix. The analysis in the Appendix implies the following lemma, which thus completes the running time analysis of the algorithm.

Lemma 4.1 *(i) A rectangle in $\mathbb{P} \cup \tilde{\mathbb{C}} \cup \mathcal{R}$ can be inserted into or deleted from $\mathbb{T}$ in $O(\log n)$ time. (ii) A query can be answered in $O(\log n)$ time.*

References

[1] http://en.wikipedia.org/wiki/Klee's_measure_problem

[2] D. Attali and H. Edelsbrunner, Inclusion-exclusion formulas from independent complexes, *Proc. 21st Ann. Sympos. Comput. Geom.*, 2005, 247–254.

[3] M. de Berg, M. van Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, 2nd edition, Springer Verlag, Heidelberg, 2000.

[4] J. L. Bentley, Algorithms for Klee's rectangle problems. Unpublished notes, Computer Science Department, Carnegie Mellon University, 1977.

[5] J.D. Boissonnat, M. Sharir, B. Tagansky and M. Yvinec, Voronoi diagrams in higher dimensions under certain polyhedral distance functions, *Discrete Comput. Geom.* 19 (1998), 485–519.

[6] E. Chen and T. M. Chan, Space-efficient algorithms for Klee's measure problem, *Proc. 17th Canadian Conf. Comput. Geom.*, 2005.

[7] B. S. Chlebus, On the Klee's measure problem in small dimensions, *Proc. 25th Conf. Current Trends in Theory and Practice of Informatics*, 1998, 304-311.

[8] H. Edelsbrunner, The union of balls and its dual shape, *Discrete Comput. Geom.* 13 (1995), 415–440.

[9] M. L. Fredman and B. Weide, The complexity of computing the measure of $\bigcup[a_i, b_i]$, *Commun. ACM* 21 (1978), 540–544.

[10] J. van Leeuwen and D. Wood, The measure problem for rectangular ranges in d-space, *J. Algorithms* 2 (1981), 282–300.

[11] V. Klee, Can the measure of $\bigcup[a_i, b_i]$ be computed in less than $O(n \log n)$ steps? *Amer. Math. Monthly* 84 (1977), 284–285.

[12] M. Overmars and C.K. Yap, New upper bounds in Klee's measure problem, *SIAM J. Comput.* 20 (1991), 1034–1045.

Appendix

A Updates and Queries to the Segment Tree

Inserting/Deleting a pillar. Let S be a pillar that we wish to insert into $\mathcal{S}$ or delete from $\mathcal{S}$. If the line $(\star)$ is executed at v, then $S \in \mathbb{P}_v$. If we are inserting S, we set $\pi(v) = \ell_v$, $\varphi(v) = \lambda_f(v) = \lambda_c(v) = \bar{\xi}_i(v) = 0$ and increment $|\mathbb{P}_v|$. If we are deleting S, we decrement $|\mathbb{P}_v|$. If $|\mathbb{P}_v| > 0$, there is nothing else to be done, so assume that $|\mathbb{P}_v|$ becomes zero. If v is a leaf, we invoke UPDATEAUX(v) to compute the auxiliary data stored at v. If v is an internal node, we first call the subroutines ADJUSTFLOOR, ADJUSTCEILING at the children of v and then invoke UPDATEAUX(v).

Inserting/Deleting a floater rectangle. Suppose we wish to insert or delete a rectangle $R \in \mathcal{R}$. If the line $(\star)$ is executed at v, then $R \in \mathcal{R}_v$. We perform the following steps at v. Let $\rho_v = \delta_v \times [\mathrm{fl}, \mathrm{cl}]$ and $r = \mathrm{Ht}(R \cap \rho_v)$.

If we are inserting R, we set $h(v) = h(v) + r$, $\varphi(v) = \varphi(v) + (\ell_v - \pi(v))r$, and set $\lambda_f(v)$ (resp., $\lambda_c(v)$) to $\ell_v - \pi(v)$, provided R is a rectangle of $\mathbb{S}^-$ (resp., $\mathbb{S}^+$). (Note that the values of $h(v)$, $\varphi(v)$ in the right-hand sides are correct because the ADJUSTFLOOR and ADJUSTCEILING subroutines have just been called at v.) On the other hand, if we are deleting R, we set $h(v) = h(v) - r$, $\varphi(v) = \varphi(v) - (\ell_v - \pi(v))r$, and set $\lambda_f(v)$ (resp., $\lambda_c(v)$) to zero, provided R is a rectangle of $\mathbb{S}^-$ (resp., $\mathbb{S}^+$). The latter action is justified by noting that if $R \in \mathbb{S}^-$ then no other rectangle of $\mathcal{R}_v^*$ belongs to $\mathbb{S}^-$, and the same is true for $\mathbb{S}^+$.

Inserting/Deleting a corner rectangle. Suppose we wish to insert or delete a rectangle $C \in \tilde{\mathbb{C}}_i$. Let χ be the y-coordinate of the horizontal edge that lies in the interior of $\square$. If line $(\star)$ is executed for v, then $C \in \tilde{\mathbb{C}}_{iv}$. We set $\bar{\xi}_i(v) = (\ell_v - \pi(v))\,\mathrm{Ht}(C)$ and $J_i(v) = [\chi, \chi]$.

Answering queries. Recall that for a y-interval Δ, W_Δ denotes the rectangle $[x_0, x_1] \times \Delta$, and $\xi(\Delta) = \mathrm{Area}[\mathcal{U}(\tilde{\mathbb{C}}) \setminus \mathcal{U}(\mathbb{P})] \cap W_\Delta$. We have to answer queries of the form: Given a y-interval $\Delta \subseteq [y_0, y_1]$, return $\xi(\Delta)$. To answer such queries we define another kind of queries as follows. Given an x-interval $\gamma \subseteq [x_0, x_1]$, return $\pi(\gamma)$, where $\pi(\gamma)$ denotes the length of the portion of γ covered by the x-projections of the pillars in $\mathbb{P}$; (note that $\pi([x_0, x_1]) = \pi$.) We refer to queries of the second type as Q1 queries and to queries of the first type as Q2 queries. See Figure 7.

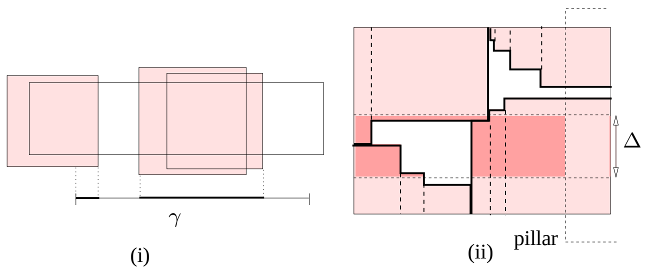

Figure 7: (i) A Q1 query; $\pi(\gamma)$ is the length of the highlighted portion of γ. (ii) A Q2 query; the shaded region is $\mathcal{U}(\mathbb{C})$, and the darkly shaded region is $\xi(\Delta)$.

Answering a Q1 query. Let I be a query interval for which we wish to compute $\pi(I)$, the length of the portion of I covered by the x-projections of the pillars. The recursive procedure described in Figure 8(i) computes $\pi(I)$. It is the standard 1-dimensional range-searching procedure, with one caveat: the endpoints of I may lie in the interior of the intervals associated with the corresponding leaves of $\mathbb{T}$, so additional (though obvious) actions are required at those leaves. The correctness is straightforward, and the running time is obviously $O(\log n)$.

```
Q1-QUERY($v$, $I$)

  if $I = \emptyset$     return 0
  if $\delta_v \subseteq I$    return $\pi(v)$
  if $I \subseteq \delta_v$ and $|\mathbb{P}_v| \geq 1$    return $\|I\|$
  returnQ1-QUERY(L($v$), $I \cap \delta_{\mathrm{L}(v)}$)
          + Q1-QUERY(R($v$), $I \cap \delta_{\mathrm{R}(v)}$)
```

(i)

```
Q2-QUERY($v$, $\Delta$)

  if $\Delta \cap J_0(v) = \emptyset$    return 0
  if $J_0(v) \subseteq \Delta$    return $\bar{\xi}_0(v)$
  returnQ2-QUERY(L($v$), $\Delta$)
          + Q2-QUERY(R($v$), $\Delta$)
```

(ii)

Figure 8: (i) The recursive procedure for answering a Q1 query; $\|I\|$ is the length of the interval I. (ii) The recursive procedure to compute $\mathrm{Area}(\mathcal{U}(\tilde{\mathbb{C}}(\Delta)) \setminus \mathcal{U}(\mathbb{P}))$.

Answering a Q2 query. Let $\Delta = [\alpha, \beta]$ be a y-interval. We describe how to compute $\xi(\Delta)$, the area of $[\mathcal{U}(\tilde{\mathbb{C}}) \setminus \mathcal{U}(\mathbb{P})] \cap W_\Delta$. Let $\xi_i(\Delta) = \mathrm{Area}([\mathcal{U}(\tilde{\mathbb{C}}_i) \setminus \mathcal{U}(\mathbb{P})] \cap W_\Delta)$, then $\xi(\Delta) = \sum_{i=0}^{3} \xi_i(\Delta)$. We describe how to compute $\xi_0(\Delta)$; the other $\xi_i(\Delta)$'s can be computed in a similar manner.

Let $\tilde{\mathbb{C}}_0(\Delta) \subseteq \tilde{\mathbb{C}}_0$ be the set of rectangles whose top edges lie in the y-interval Δ; see Figure 9. $\tilde{\mathbb{C}}_0(\Delta)$ is a contiguous subsequence of $\tilde{\mathbb{C}}_0$, and $\mathcal{U}(\tilde{\mathbb{C}}_0(\Delta))$ is also a staircase polygon $P_0(\Delta)$. Let x_L (resp., x_R) be the x-coordinate of the left (resp., right) boundary of $P_0(\Delta)$. Set $I_L = [x_0, x_L]$, $I_B = [x_L, x_R]$, $R_L = I_L \times \Delta$, and $R_B = I_B \times [y_0, \alpha]$ (see Figure 9 (ii)). Then

$$\mathcal{U}(\tilde{\mathbb{C}}_0) \cap W_\Delta = [P_0(\Delta) \setminus R_B] \cup R_L.$$

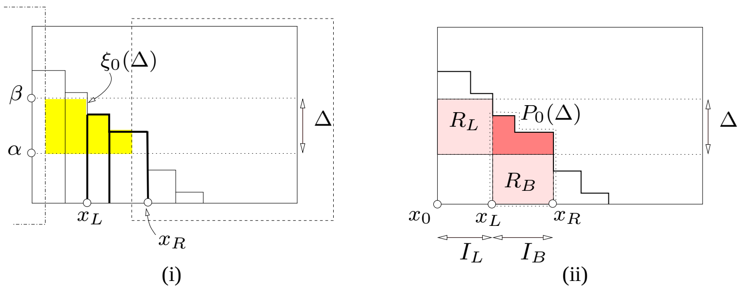

Figure 9: (i) Shaded region is $\xi_0(\Delta)$, rectangles in $\tilde{\mathbb{C}}_0(\Delta)$ are drawn in thick lines. (ii) Rectangles R_B and R_L.

Since R_L and $P_0(\Delta)$ are disjoint and $R_B \subseteq P_0(\Delta)$, we have

$$\begin{aligned}\xi_0(\Delta) &= \mathrm{Area}(P_0(\Delta) \setminus \mathcal{U}(\mathbb{P})) - \mathrm{Area}(R_B \setminus \mathcal{U}(\mathbb{P})) + \mathrm{Area}(R_L \setminus \mathcal{U}(\mathbb{P})) \\ &= \mathrm{Area}(P_0(\Delta) \setminus \mathcal{U}(\mathbb{P})) - (x_R - x_L - \pi(I_B))(\alpha - y_0) + (x_L - x_0 - \pi(I_L))(\beta - \alpha).\end{aligned}$$

We can compute $\pi(I_B)$ and $\pi(I_L)$ by performing Q1 queries with I_B and I_L. The first term is computed using the recursive procedure Q2-QUERY, described in Figure 8 (ii). The running time of Q2-QUERY is

$O(\log n)$ because the intervals $J_0(v)$ are ordered along the leaves of $\mathbb{T}$ and the parents of the nodes visited by the procedure lie on two paths of $\mathbb{T}$.

Putting everything together, we obtain the following lemma and Lemma 4.1.

Lemma A.1 *(i) A rectangle in $\mathbb{P} \cup \tilde{\mathbb{C}} \cup \mathcal{R}$ can be inserted into or deleted from $\mathbb{T}$ in $O(\log n)$ time. (ii) A Q1 or a Q2 query can be answered in $O(\log n)$ time.*

This lemma provides the missing ingredients for the algorithm described in Section 3, and thus, at long last, completes the proof of Theorem 1.1.