A Discrete and Dynamic Version of Klee's Measure Problem

Hakan Yıldız* John Hershberger† Subhash Suri*

Abstract

Given a set of axis-aligned boxes $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$ and a set of points $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$ in d-space, let the *discrete measure* of $\mathcal{B}$ with respect to $\mathcal{P}$ be defined as $meas(\mathcal{B}, \mathcal{P}) = |\mathcal{P} \cap \{\bigcup_{i=1}^{n} B_i\}|$, namely, the number of points of $\mathcal{P}$ contained in the union of boxes of $\mathcal{B}$. This is a discrete and dynamic version of Klee's measure problem, which asks for the *Euclidean* volume of a union of boxes. Our result is a data structure for maintaining $meas(\mathcal{B}, \mathcal{P})$ under dynamic updates to both $\mathcal{P}$ and $\mathcal{B}$, with $O(\log^d n + m^{1-\frac{1}{d}})$ time for each insert or delete operation in $\mathcal{B}$, $O(\log^d n + \log m)$ time for each insert and $O(\log m)$ time for each delete operation in $\mathcal{P}$, and $O(1)$ time for the measure query. Our bound is slightly better than what can be achieved by applying a more general technique of Chan [3], but the primary appeal is that the method is simpler and more direct.

1 Introduction

A classical problem in computational geometry, known as Klee's Measure Problem, asks for an efficient algorithm to compute the volume of the union of n axis-aligned boxes in d dimensions. While optimal $O(n \log n)$ time algorithms are known for dimensions one and two [1, 7], the best bound in higher dimensions is roughly $O(n^{d/2})$ [4]. Indeed, despite more than twenty years of effort, the barrier of $O(n^{3/2})$ remains unbroken even in three dimensions. It is known, however, that as the dimension becomes large, the problem is NP–hard [2].

In this paper, we consider a *discrete* and *dynamic* version of Klee's problem, in which the volume of a box is defined as the cardinality of its intersection with a finite point set $\mathcal{P}$, and both the boxes and the points are subject to insertion and deletion. In particular, we have a set of axis-aligned boxes $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$, a set of points $\mathcal{P} = \{p_1, p_2, \ldots, p_m\}$ in d-space, and we wish to maintain the *discrete measure* of $\mathcal{B}$ with respect to $\mathcal{P}$, namely, $meas(\mathcal{B}, \mathcal{P}) = |\mathcal{P} \cap \{\bigcup_{i=1}^{n} B_i\}|$, under insertion and deletion of both points and boxes.

The problem is fundamental, and arises naturally in several applications dealing with multi-attribute data. In databases, for instances, data records with d independent attributes are viewed as d-dimensional points, and selection rules are given as ranges over these attributes. A conjunction of ranges over d attributes is then equivalent to a d-dimensional box. Given a set of selection rules, the problem of *counting* all the data records that satisfy the union (namely, the disjunction) of all the rules is our discrete measure problem. Similarly, one may ask for the set of records that fail to satisfy *any of the rules*, and thus form the set of points "not covered" by the union of boxes.

Similarly, the management of firewall rules for network access can also be formulated as a discrete measure problem. The data packets in the Internet are classified by a small number of fields, such as IP address of the source and destination, the network port number, etc. The managers of a local area network (LAN) use a number of "firewall rules" based on these attributes to block some external services (such as ftp) from their network. The discrete measure problem in this setting keeps track of the number of services blocked by all the firewall rules; conversely, one can keep track of the number of services that become "exposed" by the deletion of a box.

Problem Formulation and Our Results

We begin with a formal definition of the problem. A d-dimensional *box* B is the Cartesian product of d one-dimensional ranges, namely $B = \prod_{i=1}^{d} [a_i, b_i]$, where a_i and b_i are reals. The *discrete measure* of a single box B with respect to a finite set of points $\mathcal{P}$ is the cardinality of the intersection $\mathcal{P} \cap B$. The discrete measure of the set of boxes $\mathcal{B}$ with respect to $\mathcal{P}$, denoted $meas(\mathcal{B}, \mathcal{P})$, is the cardinality of $\mathcal{P} \cap \{\bigcup_{B \in \mathcal{B}} B\}$. (Because a point may lie in multiple boxes, the discrete measure of $\mathcal{B}$ is not the sum of the measures of the individual boxes.) In this paper, we consider the problem of maintaining the discrete measure under insertion and deletion of both points and boxes. Specifically, we propose a data structure that supports modifying $\mathcal{P}$ through insertion or deletion of a point, modifying $\mathcal{B}$ through insertion or deletion of a box, and querying for the current discrete measure $meas(\mathcal{B}, \mathcal{P})$.

Despite its natural formulation, the problem appears not to have been studied in this form. This may be partially attributed to the fact that the *static* version of the problem is easy to solve using standard

*Department of Computer Science, University of California, Santa Barbara, {hakan,suri}@cs.ucsb.edu

†Mentor Graphics Corp., john_hershberger@mentor.com

data structures of computational geometry: build a d-dimensional version of a segment tree for the set of boxes, and then query separately for each point to determine whether any box contains it, for a total of $O(n \log^d n + m \log^{d-1} n)$ time. This approach, however, is inefficient when the set of boxes is dynamic, because each insertion or deletion can affect a large number of points, requiring $\Omega(m)$ recomputation time per update.

During the writing of this paper, we discovered that a technique of Chan [3] can be used to solve this problem. In [3], he describes a data structure for maintaining a set of points and a set of hyperplanes in d-space to answer queries of the form "does any of the points lie below the lower envelope of the hyperplanes." One can use this data structure in combination with standard range searching structures and a dynamization technique by Overmars and van Leeuwen [9] to solve our discrete measure problem so that point insertions and box updates require $O(\log^2 m + \log^d n)$ and $O(m^{1-\frac{1}{d}} \log m + \log^d n)$ time respectively.[1]

Compared to this bound, the time complexity of our data structure is better by a factor of $\log m$. However, a more important contribution may be the simplicity of our method and the fact that it solves the problem in a more *direct* way, making it more appealing for implementation. Specifically, our result gives a dynamic data structure for the discrete measure problem with the following performance: a box can be inserted or deleted in time $O(m^{1-\frac{1}{d}} + \log^d n)$; a point can be inserted in time $O(\log m + \log^d n)$ and deleted in time $O(\log m)$. The data structure always updates its measure, so a query takes $O(1)$ time.

The data structure also solves the *reporting* problem in output-sensitive time. Specifically, if k is the number of points in the union of the boxes, then they can be found in $O(k + k \log \frac{m}{k})$ worst-case time. The same bound also holds if one wants to report the points *not* contained in the union. Finally, we extend our results to a *stochastic* version of the problem, in which each point and each box is associated with an independent probability of being present. In this case, one can naturally define an *expected discrete measure*, which is the expected number of points present that are covered by the union of the boxes present. Our bounds for the stochastic case are the same as the deterministic one.

2 Maintaining the Discrete Measure

In the following discussion we assume that all the boxes in $\mathcal{B}$ and points in $\mathcal{P}$ have distinct coordinates.[2] Before we describe our dynamic data structure, it is helpful to consider a solution for the static problem. Let $\mathcal{B}$ be a set of n boxes and $\mathcal{P}$ a set of m points in d-space. For each point $p \in \mathcal{P}$, we define its *stabbing count*, denoted $stab(p)$, as the number of boxes in $\mathcal{B}$ that contain p. The *measure* of a single point p, $meas(\mathcal{B}, \{p\})$, is 1 if $stab(p) > 0$ and 0 otherwise. One can easily see that the overall discrete measure can be written as the sum of point measures; that is, $meas(\mathcal{B}, \mathcal{P}) = \sum_{p \in \mathcal{P}} meas(\mathcal{B}, \{p\})$. The stabbing count of a point can be efficiently obtained using a *multi-level segment tree* [10], which achieves the following performance bounds.

Lemma 1 ([10]) *The multi-level segment tree represents a set of n boxes in d-space. The structure can report the stabbing count of any query point in $O(\log^{d-1} n)$ time. It requires $O(n \log^{d-1} n)$ space and $O(n \log^d n)$ preprocessing time for construction.*

By building a multi-level segment tree and then querying it for the stabbing count of each point in $\mathcal{P}$, we can calculate the measure $meas(\mathcal{B}, \mathcal{P})$ for the static problem in $O(n \log^d n + m \log^{d-1} n)$ time using $O(n \log^{d-1} n)$ space.

2.1 Invariants for Stabbing and Measure

The static solution described above loses its appeal in the dynamic setting because each box insertion or deletion can invalidate the stabbing count of $\Omega(m)$ points. We circumvent this problem by storing the stabbing counts *indirectly*, using an idea from *anonymous segment trees* [12], so that only a small number of these indirect values need to be modified after a box update. We describe the technique in general first, deferring its specialization for the efficient maintenance of the discrete measure until later.

Consider a balanced tree (not necessarily binary) whose leaves are in one-to-one correspondence with the points of $\mathcal{P}$. The point corresponding to a leaf v is denoted p_v. In order to represent the stabbing counts of the points, we store a *non-negative* integer field $\sigma(w)$ at each node w of the tree subject to the following *sum invariant*: *for each leaf v, the sum of $\sigma(a)$ over all ancestors a of v (including v itself) equals $stab(p_v)$*. By assigning $\sigma(v) = stab(p_v)$ to each leaf v and $\sigma(w) = 0$ to all internal nodes w, we may obtain a trivial assignment with the sum invariant. But, as we will see, the flexibility afforded by these σ values allows us to update the stabbing counts of many points by modifying only a few σ values. As an example, if a box covering all the points of $\mathcal{P}$ were inserted, then incrementing the single value $\sigma(root)$ by 1 suffices, where *root* denotes the root of the tree.

We will maintain the discrete measure, $meas(\mathcal{B}, \mathcal{P})$, through the σ values. In particular, at each node v, we

[1] Reducing box update time is possible at the expense of increasing the cost of point insertions and vice versa.

[2] This assumption merely simplifies the presentation; one can use symbolic perturbation to break ties between identical coordinates without affecting the result.

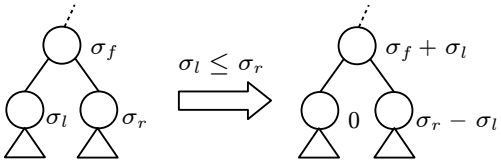

Figure 1: The push-up operation on a node with two children.

store a quantity $\bar{\mu}(v)$ representing the number of points that have a stabbing count of 0, considering only the information stored in the subtree rooted at v. (The notation $\bar{\mu}$ is meant to suggest that it represents the complement of the measure.) The quantity $\bar{\mu}(v)$ is defined recursively using the σ values as follows:

$$\bar{\mu}(v) = \begin{cases} 0 & \text{if } \sigma(v) > 0 \\ 1 & \text{if } \sigma(v) = 0 \ \wedge \ v \text{ is a leaf} \\ \sum_{w \in child(v)} \bar{\mu}(w) & \text{if } \sigma(v) = 0 \ \wedge \\ & \quad v \text{ is an internal node} \end{cases}$$

where $child(v)$ represents the children of a non-leaf node v. It is easy to show that $\bar{\mu}(root)$ is the number of points in $\mathcal{P}$ whose stabbing counts are 0. Consequently, $meas(\mathcal{B}, \mathcal{P}) = m - \bar{\mu}(root)$, and one can report $meas(\mathcal{B}, \mathcal{P})$ in $O(1)$ time.

We add one final constraint on σ values to achieve uniqueness, which also contributes to the efficiency of our specialized structure. In particular, we push the σ values as high up the tree as possible to enforce the following *push-up invariant*: *at least one child of every non-leaf node v has a σ value of* 0. This specifies σ uniquely, as shown by the following lemma.

Lemma 2 *Let T be a tree representing a set of points $\mathcal{P}$ and their stabbing counts as described above. Then there exists a unique configuration of σ values satisfying the sum and the push-up invariants in T.*

Proof. We prove only the existence of the desired configuration due to space limitation; the proof of uniqueness can be found in the full version of the paper. Consider an arbitrary configuration of σ values satisfying the sum invariant. (For instance, $\sigma(v) = stab(p_v)$ for each leaf and $\sigma(v) = 0$ for each non-leaf.) We then apply the following *push-up* operation at each non-leaf node v to revise its value: increment $\sigma(v)$ by Δ and decrement $\sigma(w)$ by Δ for each child w of v, where $\Delta = \min_{w \in child(v)} \sigma(w)$. (See Figure 1). This achieves the push-up invariant at v while preserving the sum invariant in the tree. Repeated applications of the push-up operation from the leaves to the root produce a configuration of σ values satisfying both invariants.

□

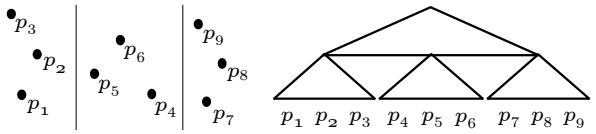

Figure 2: A measure tree of 9 points on the plane.

2.2 The Measure Tree and Dynamic Updates

In order to allow efficient insertion and deletion of boxes, and the corresponding updates of the points' stabbing counts, we organize $\mathcal{P}$ in a balanced tree that supports efficient range queries. A k-d tree, where points are stored at the leaves, allows efficient range queries, but is inefficient for insertion and deletion of *points*.[3] The structure we propose, which we call a *measure tree*, is a variant of divided k-d trees [11], and allows both efficient range queries and updates on the set of points. We note that the tree described in this section has slightly slower amortized bounds but these can be easily improved to achieve our main result as explained in Section 2.5.

We describe the measure tree in two dimensions for simplicity; the extension to d dimensions is conceptually straightforward, but we defer those details for later. Given a dynamic set of points $\mathcal{P}$ in the plane, we represent $\mathcal{P}$ as a two-level tree. The first level consists of an upper tree that partitions the points of $\mathcal{P}$ into at most $2\sqrt{m}$ subsets along the x-axis, each containing at most $2\sqrt{m}$ points, where m is the current size of $\mathcal{P}$. Each leaf of the upper tree acts as a root for a lower level tree that further partitions the corresponding subset of points using their y-coordinates. These lower trees form the second level of our tree. Figure 2 shows an example. Both levels of the tree are organized using 2-3 trees in which each data element is stored in a single leaf. Consequently, each leaf of the measure tree corresponds to a single point of $\mathcal{P}$ and we can use our measure maintenance scheme to store σ and $\bar{\mu}$ values on the nodes. We now discuss how to perform updates on the measure tree while preserving the invariants.

Insertion or Deletion of a box B. Let us consider insertion first. We find a set $\mathcal{C}$ of subtrees whose leaves correspond to the points covered by B. This is a range query, where we first perform a one-dimensional range search on the upper tree to locate the subsets of points that are completely or partially covered by the x-range of B. Observe that at most two subsets are partially covered. We then search the lower level trees corresponding to the partially covered subsets to find the points contained in B. The leaves corresponding to these points are included in $\mathcal{C}$. For each subset that is completely covered by the x-range of B, we perform

[3] There is also no easy way to implement our scheme using range trees because they contain multiple copies of the points.

a one-dimensional range search on the corresponding lower tree to find a set of maximal subtrees containing the points that lie in B. These maximal subtrees are also included in $\mathcal{C}$. It is straightforward to show that the subtrees in $\mathcal{C}$ span the set of points covered by B and the total cost of the range query is $O(\sqrt{m}\log m)$.

The insertion of B causes the stabbing count of each point contained in B to increase by 1. We effect this by incrementing the σ value of the root of each subtree in $\mathcal{C}$ by 1. This corrects the sum invariant in the tree, but may invalidate the push-up invariant. We therefore apply push-ups on the nodes whose σ values are updated. Since each push-up may introduce a violation of the push-up invariant at the parent, we continue applying push-ups until all violations are resolved. Finally, we recompute $\bar{\mu}$ for all ancestors of nodes whose σ values changed. This recomputation is also done bottom-up, since the $\bar{\mu}$ value of a node depends on the $\bar{\mu}$ values of its descendants. We note that both the push-ups and the recomputations of $\bar{\mu}$ values can be done as part of the tree traversal of the range query. It follows that the total cost of the box insertion is $O(\sqrt{m}\log m)$ time.

The handling of deletion is similar to insertion, except that we decrement the σ value of the root of each subtree found by the range query. The time complexity is $O(\sqrt{m}\log m)$, as for insertion. Decrementing the σ's may cause some values to drop below zero, but the push-up operations eliminate these negative values. In particular, observe that a push-up at a node v restores not only the push-up invariant but also the non-negativity of v's children. To see that the final value of $\sigma(root)$ is non-negative, imagine a root-to-leaf path (as in the proof of uniqueness for Lemma 2 found in the appendix) such that σ is zero for all nodes on the path except *root*. The path ends at a leaf v such that $stab(p_v)$ equals $\sigma(root)$, and so it follows that $\sigma(root)$ is non-negative.

Insertion or Deletion of a point *p*. When inserting a point p, we search the upper tree with the x-coordinate of p to find the lower tree in which p should be inserted, and then insert p using the standard 2-3 tree insertion algorithm. This creates a new leaf v with $p_v = p$. We need to know the stabbing count of p in order to initialize $\sigma(v)$ correctly. For the moment, let us assume that we know $stab(p)$—see Lemma 3—and focus on the update of the tree. In order to preserve the sum invariant, we set $\sigma(v)$ to $stab(p) - \Sigma$, where Σ is the sum of $\sigma(a)$ over all strict ancestors a of v. If $\sigma(v)$ is less than 0, we apply push-ups to v and all of its ancestors to push the negativity to the root, where it is canceled out. The 2-3 tree insertion may split one or more ancestors of v, and during those splits, the σ values of the resulting nodes are set to the original node's σ value, thereby preserving the sum invariant. After the split, we apply push-ups on the resulting nodes to re-establish the push-up invariant. Altogether, $O(\log m)$ splits and push-ups are performed, and so the cost of the insertion is $O(\log m)$. The insertion might cause the size of the lower tree to exceed $2\sqrt{m}$, but this is discussed in Section 2.3.

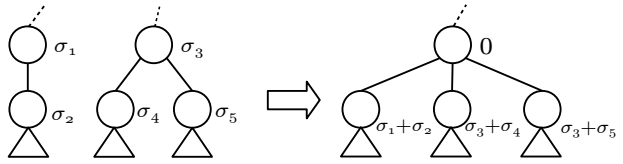

Figure 3: Push-down in a merge.

When a point p is deleted, we locate the lower tree containing it and simply delete the leaf corresponding to p, and restructure the tree to reestablish the balance of the 2-3 tree. The sum invariant is unaffected by the deletion, but we may need to apply push-ups to the ancestors of v to restore the push-up invariant. The deletion may also cause 2-3 tree merge or redistribution operations, and to preserve the sum invariant during these operations, we push the σ values of the participating nodes down to their children (see Figure 3). After the operation, push-ups are applied on these nodes to restore the push-up invariant. If the lower tree becomes empty as a result of the deletion, we simply delete it and apply the same deletion algorithm on the upper tree. Due to the decrease in the value of m, the sizes of some upper or lower trees may exceed $2\sqrt{m}$; we deal with this in Section 2.3.

2.3 Complexity Analysis

We use two types of operations to ensure that the upper and the lower trees do not exceed their size thresholds. First, when a lower tree's size exceeds $2\sqrt{m}$, we split it into two new lower trees of equal size, destroying the original tree and constructing the new trees from scratch. During this process, we traverse the original tree to obtain the stabbing counts of the points and use those to construct the new trees. This split operation takes $O(\sqrt{m})$ time since the y-order of the points is known. Second, we avoid violating the upper tree's threshold by periodically rebuilding the entire measure tree. This reconstruction creates a lower tree for each set of $\lceil\sqrt{m}\,\rceil$ points along the x-axis (except perhaps the last one in the sequence, which may be smaller). Consequently, the size of the upper tree is at most $\sqrt{m}$. The reconstruction takes $O(m\log m)$ time. (It can be done in $O(m)$ time if we maintain the x- and y-orders of the points separately.) We determine when to do the reconstruction as follows. Assume that the most recent reconstruction of the tree was done when $m = m_0$. We reconstruct the tree after $\frac{1}{5}m_0$ point insertions or deletions. This ensures that the upper tree does not exceed its threshold. The proof is straightforward and left to

the reader.

Next, we discuss how to initialize the stabbing count of a point when it is first inserted. We enable this by maintaining a separate *dynamic multi-level segment tree* [5], which provides the following functions dynamically.

Lemma 3 ([5]) *The dynamic multi-level segment tree represents a dynamic set of n boxes in d-space. The structure uses $O(n \log^{d-1} n)$ space and can report the stabbing count of any query point in $O(\log^d n)$ time. It supports insertion or deletion of boxes in $O(\log^d n)$ time apiece.*

Putting together these pieces, we obtain our main result in two dimensions.

Theorem 4 *We can maintain the discrete measure in two dimensions using $O(n \log n + m)$ space, with constant time measure queries, $O(\log^2 n + \sqrt{m} \log m)$ time for insertion or deletion of a box, $O(\log^2 n + \log m)$ time for a point insertion, and $O(\log m)$ time for a point deletion time. (The $\log m$ term in the bounds is amortized.)*

Proof. We use the measure tree along with a two-dimensional dynamic segment tree. The bound on the space complexity follows because the measure tree requires linear space and the multi-level segment tree requires $O(n \log n)$ space by Lemma 3. The query complexity is obviously constant. The insertion or deletion of a box takes $O(\sqrt{m} \log m)$ time for the measure tree and $O(\log^2 n)$ time for the segment tree. The cost of inserting or deleting a point is $O(\log m)$ for the measure tree if there is no reconstruction of a lower tree or the whole measure tree. Reconstruction of the measure tree costs $O(m \log m)$. We charge the cost of this construction to the $\Omega(m)$ point updates that must precede it, which gives us an amortized cost of $O(\log m)$ per update. The reconstruction of a lower tree costs $O(\sqrt{m})$. One can easily show that $\Omega(\sqrt{m})$ point insertions precede the construction, which gives us an amortized cost of $O(1)$. Finally, we do a stabbing count query costing $O(\log^2 n)$ time when we insert a point. This completes the proof. □

2.4 Extension to Higher Dimensions

The measure tree naturally extends to higher dimensions, as a d-level tree, with each level partitioning the points along one of the coordinate axes. The tree at the top level partitions the set of points into at most $2m^{1/d}$ subsets, each of which is partitioned into at most $2m^{1/d}$ subsets by a second level tree. This partitioning continues through d levels. The measure is maintained using the σ and $\bar{\mu}$ values, as in two dimensions.

All the update procedures are natural extensions of their two-dimensional counterparts. The initial tree size is at most $\lceil m^{1/d} \rceil$ at all levels; a tree is split when its size becomes larger than $2m^{1/d}$. Moreover, one can show that there is a positive constant C such that reconstructing the tree after each Cm_0 point insertions or deletions guarantees that the size of the upper tree is bounded by $2m^{1/d}$. The following theorem summarizes the bounds of the d-dimensional structure; its proof is similar to that of Theorem 4.

Theorem 5 *We can maintain the discrete measure in d dimensions, for $d \geq 2$, using $O(n \log^{d-1} n + m)$ space, with constant time measure queries, $O(\log^d n + m^{1-\frac{1}{d}} \log m)$ time for insertion or deletion of a box, $O(\log^d n + \log m)$ time for insertion of a point, and $O(\log m)$ time for the deletion of a point. (The $\log m$ term in the bounds is amortized.)*

2.5 Further Improvements

The amortized bounds of our structure can be converted to worst case bounds, using a technique called *global rebuilding* [8]. The idea, in brief, is to spread out the process of subtree reconstruction over time, operating on a shadow copy of the main data structure and then swapping in the result when the reconstruction is finished.

Finally, the term $m^{1-\frac{1}{d}} \log m$ in box update bounds can be improved to $m^{1-\frac{1}{d}}$ by using an optimized version of the measure tree. For instance, in two dimensions, the partitioning parameter can be tuned to achieve $O(\sqrt{m} + \log^2 n)$ time for inserting or deleting a box, by mimicking the construction of [6].

3 Extensions

3.1 Reporting Queries

In some applications, it is useful to report explicitly the points covered (or uncovered) by the union of boxes. Our structure can be used to answer such queries in output-sensitive time as in the following theorem.

Theorem 6 *A reporting query can be answered in $O(k + k \log \frac{m}{k})$ time, where k is the size of the output.*

Proof. The proof will appear in the full version of the paper. □

3.2 Stochastic Discrete Measure

The recent proliferation of data mining applications has created an urgent need to deal with *data uncertainty*, which may arise because the mining algorithms output probability distributions over an output space, or because attributes whose values are not explicitly known are modeled with a discrete set of probabilistic values.

This motivates a natural *stochastic* extension of our discrete measure problem, in which both the underlying set of points $\mathcal{P}$ and the set of boxes $\mathcal{B}$ are associated with independent probabilities. Specifically, each point p in $\mathcal{P}$ occurs with probability π_p and each box B in $\mathcal{B}$ occurs with probability π_B. The probabilities are independent, but otherwise can take any real values. A natural problem in this setting is to compute the *expected size* of the discrete measure—that is, how large is $meas(\mathcal{B}, \mathcal{P})$ for a random sample of boxes and points drawn from the given probability distribution?

Our structure can be easily adapted to this stochastic problem with the same complexity bounds. The details will appear in a journal version of the paper.

Theorem 7 *The d-dimensional stochastic measure problem can be solved with a data structure that requires $O(n \log^{d-1} n + m)$ space, $O(1)$ query time, $O(\log^d n + m^{1-\frac{1}{d}})$ time for insertion or deletion of a box, $O(\log^d n + \log m)$ time for a point insertion and $O(\log m)$ time for a point deletion.*

4 Closing Remarks

We introduced a discrete measure problem, and presented a data structure that supports dynamic updates to both the set of points and the set of boxes. The queries for the current measure take constant time, the updates to the set of points take polylogarithmic time, while updates to the set of boxes take time polylogarithmic in the number of boxes and sub-linear in the number of points. The data structure permits output-sensitive enumeration of the points covered by the union of the boxes, and also lends itself to a stochastic setting in which points and boxes are present with independent, but arbitrary, probabilities.

Our work leads to a number of research problems. First, can the update bounds be improved? Second, is there a trade-off between the update time for boxes and the update time for points? In particular, can one achieve polylogarithmic complexity in both n and m?

References

[1] J. L. Bentley. Solutions to Klee's rectangle problems. *Unpublished manuscript, Dept. of Comp. Sci., CMU, Pittsburgh PA*, 1977.

[2] K. Bringmann and T. Friedrich. Approximating the volume of unions and intersections of high-dimensional geometric objects. *CGTA*, 43:601–610, 2010.

[3] T. Chan. Semi-online maintenance of geometric optima and measures. In *Proc. 13th annual ACM-SIAM Symposium on Discrete algorithms*, pages 474–483, 2002.

[4] T. M. Chan. A (slightly) faster algorithm for Klee's measure problem. *CGTA*, 43(3):243–250, 2010.

[5] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. *Report F59, Institut für Inform., TU Graz*, 1980.

[6] K. Kanth and A. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. ICDT*, page 257. Springer, 1998.

[7] V. Klee. Can the measure of $\cup[a_i, b_i]$ be computed in less than $O(n \lg n)$ steps? *American Mathematical Monthly*, pages 284–285, 1977.

[8] M. Overmars. *The design of dynamic data structures*. Springer, 1983.

[9] M. Overmars and J. van Leeuwen. Worst-case optimal insertion and deletion methods for decomposable searching problems. *Information Processing Letters*, 12(4):168–173, 1981.

[10] V. K. Vaishnavi. Computing point enclosures. *IEEE Trans. Comput.*, 31:22–29, January 1982.

[11] M. van Kreveld and M. Overmars. Divided k-d trees. *Algorithmica*, 6(1):840–858, 1991.

[12] H. Yıldız, L. Foschini, J. Hershberger, and S. Suri. The union of probabilistic boxes: Maintaining the volume. In *Proc. 19th Annual European Symposium on Algorithms (ESA)*, 2011.