Data Structures for Maintaining Set Partitions*

Michael A. Bender† Saurabh Sethia‡ Steven S. Skiena§

December 16, 2003

Abstract

Efficiently maintaining the partition induced by a set of features is an important problem in building decision-tree classifiers. In order to identify a small set of discriminating features, we need the capability of efficiently adding and removing specific features and determining the effect of these changes on the induced classification or partition.

In this paper we introduce a variety of randomized and deterministic data structures to support these operations on both general and geometrically induced set partitions. We give both Monte Carlo and Las Vegas data structures that realize near-optimal time bounds and are practical to implement. We then provide a faster solution to this problem in the geometric setting. Finally, we present a data structure that efficiently estimates the number of partitions separating elements.

Keywords: Set Partitions, Decision Trees, Randomized Algorithms, Approximation Algorithms, Data Structures, Random Walks.

1 Introduction

Each test or *feature* in a classification system defines a *set partition* on a class of objects. Adding new features refines the classification, whereas deleting features may result in merging previously distinguished classes. As an illustration, consider the set of automobile types { VW Beetle, Toyota, Lexus, Cadillac }. The feature *size* partitions the cars into sets of small and large cars, {{ VW Beetle, Toyota }, { Lexus, Cadillac }}. The feature *domestic-origin* partitions the cars into {{ VW Beetle, Toyota, Lexus }, { Cadillac }}. The feature *ugly-shape* distinguishes { VW Beetle, Cadillac } from { Toyota, Lexus }. Incorporating both *size* and *origin* induces the refined partition {{ VW Beetle, Toyota }, { Lexus }, { Cadillac }}, whereas the union of all three features completely

*This work appeared in preliminary form in *The Seventh Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 83–96, July 2000 [6].

†Department of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794-4400, USA. Email: `bender@cs.sunysb.edu`. Supported in part by HRL Laboratories, ISX Corporation, NSF Grants EIA-0112849 and CCR-0208670, and Sandia National Laboratories.

‡School of Electrical Engineering and Computer Science, Oregon State University, Corvallis, OR 97331-3202, USA. Email: `saurabh@eecs.orst.edu`.

§Department of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794-4400, USA. Email: `skiena@cs.sunysb.edu`. Supported in part by NSF grant CCR-9109289 and New York Science and Technology Foundation grants RDG-90171 and RDG-90172.

distinguishes the types of cars. In fact, *size* and *ugly-shape* are sufficient for complete identification, so *domestic-origin* could be deleted from the set of features without affecting the induced partition.

Efficiently maintaining the partition induced by a set of features is an important problem in building decision tree classifiers. For example, in building an optical character recognition (OCR) system [36, 35] based on point-probe decision trees [3], each of the 1500-plus pixels in each character-sized window of the image may be evaluated as a possible feature. An important goal is to find a small, robust set of probe points sufficient to distinguish among the 70-plus characters in a font, a process that may require repeatedly inserting and deleting features to see the impact on the final classification. See Figure 1 for an example of a point-probe decision tree.

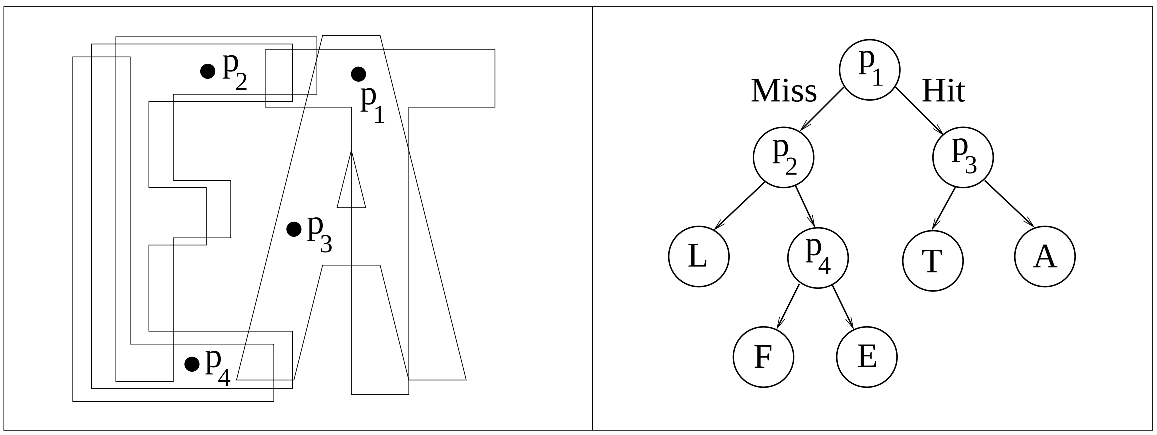

Figure 1: An example of a set S of 5 polygonal models (left) and a probe tree (right) that can determine which model is actually present.

1.1 Our Results

In this paper, we introduce techniques to speed up this process of feature identification. We propose a series of data structures for maintaining a collection of set partitions on elements $U = \{1, \ldots, n\}$. The data structures efficiently support the following three operations:

- $Insert(P, S)$ – add a new partition P to the set of partitions S.
- $Delete(P, S)$ – delete the existing partition P from the set of partitions S.
- $Report(S)$ – report the set partition of U induced by the set of partitions in S. Elements i and j are separated in the induced set partition if they are separated in at least one of the partitions in S.

We also present generalizations of the set-partition problem. Particularly interesting is the variety of algorithmic techniques that the data structures encompass, including classical balanced trees, random walks and fingerprints, word-level parallelism, and spanning trees of low stabbing number.

Our results include the following:

- We provide a data structure that supports the insert and report operations in optimal $O(n)$ worst-case time for general set partitions. Deletions take $O(n \log k)$ time, where k is the number of set partitions currently in the data structure and n is the number of elements in each partition. These results are relatively straightforward, but they provide insight into the challenges of improving the time complexity to $O(n)$.

- We provide randomized Monte Carlo and Las Vegas data structures that support all three operations on bipartitions in linear or near-linear expected time, although the Las Vegas bounds are amortized. The Monte Carlo data structure is asymptotically optimal, and the Las Vegas data structure is within a factor of $\alpha(n)$ of optimal, where $\alpha(n)$ is the inverse Ackerman function. The Monte Carlo data structure can be modified to run on general partitions without increasing the asymptotic cost, whereas the Las Vegas data structure can be modified to run on D-partitions (partitions breaking the elements into D sets) by increasing the asymptotic cost by a factor of $\log D$. Our Monte Carlo data structure is practical because of its simplicity and is used as a building block for other data structures and algorithms in this paper such as the Las Vegas data structure and the geometric data structure.

- Robust classifiers compensate for noisy features by requiring more than one piece of evidence to distinguish between every pair of objects. We provide a Monte Carlo data structure that permits us to insert/delete bipartitions efficiently and to query arbitrary pairs of elements $\{i, j\}$ to obtain the *approximate* number of bipartitions currently distinguishing i from j. Insert/delete runs in amortized time $O(n \log \log n)$ and query runs in time $O(\log n)$. Our approximation is accurate within a $(1 + \varepsilon)$ factor. The constant in the running time varies inversely with the error parameter ε. This data structure uses techniques from random walks on a line. Randomization and approximation appear to be powerful techniques in this setting, because, to our knowledge, the best known exact deterministic data structures require $O(n^2)$ time per insertion or deletion.

- We provide an efficient Monte Carlo data structure for maintaining geometric set partitions, where the set partitions are induced by linear separators of points in the plane. We achieve $O(\sqrt{n} \log n)$ time for insertion/deletion and linear report time, after an initial $O(n^{1.5} \log n)$ preprocessing step.

All our algorithms work in the standard RAM model of computation, in which machine words have $\Omega(\log n)$ bits.

This paper is organized as follows. We present deterministic data structures for maintaining set partitions in Section 2. More efficient randomized data structures are presented in Section 3. The problem of maintaining robust classifiers is discussed in Section 4. The special case of set partitions induced by geometric arrangements is discussed in Section 5.

1.2 Previous Work

A considerable literature on the subject of classification and regression trees exists; see [7, 10, 29, 34]. The abstract decision tree problem takes as input a universal set $U = \{1, \ldots, n\}$ and a family of subsets of U, $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$. Each subset defines a bipartition of U (T_i and $U - T_i$), so T represents the set of possible features. Hyafil and Rivest [24] prove that it is NP-complete to construct a minimum height or a minimum external path length decision tree. Garey [19] presents dynamic programming algorithms for determining an optimal weighted decision tree. It is NP-complete to identify the minimum set of features sufficient to build a decision tree even in very restricted geometric instances [5, 17, 18]. See [37] for a survey of related results in geometric probing.

A variety of data structures for sets and set partitions are known, including dictionaries and bit vectors, but these are not directly applicable to our problem. The primary difficulty of our problem lies in the fact that deleting a set partition may or may not result in the merger of two parts of the current induced partition, depending upon which other set partitions are included in the data structure. Union-find data structures [38] provide some support for merging disjoint subsets (as occurs on deleting a partition), but do not permit us to break up subsets (as occurs on adding a partition).

Partition refinement techniques are used in a variety of algorithms, notably minimizing deterministic finite automata [22] and its generalizations [32]. Habib et al. [21] demonstrate that partition refinement can lead to simple and efficient algorithms for graphs, strings, and matrices – although none of these operations involves deleting arbitrary set partitions. In the theory of classification, many measures of set partition similarity have been developed, the most widely used of which are the Rand index [33], the Adjusted Rand index [23], the Fowlkes and Mallows measure [16], and the Jaccard index [25]. Problems on finding the median or consensus set partition under such measures have also been studied [4, 39].

Yellin [41] efficiently supports a variety of subset testing operations (insert/delete elements, create subsets, subset and intersection queries) in $O(n^{1/2} \log n)$ time per operation, but it is not clear how to use these operations to improve even the naïve bounds for our problem. Further, near matching lower bounds are known on the complexity of any data structure that supports these operations [13].

The problem of maintaining induced set partitions can be reduced to updating an *ambiguity graph* on the n elements, where the presence of edge (i, j) indicates that elements i and j occur in different parts of at least one of the k partitions. An extensive literature exists on efficient dynamic graph algorithms [14], for such tasks as maintaining connected components under edge insertion and deletion. However, the insertion or deletion of a single n-element set partition can effect the status of $\Theta(n^2)$ edges in such an ambiguity graph, rendering such an approach infeasible.

The most closely related work appears in an unpublished manuscript of Calinescu [8]. Calinescu proposes an alternative data structure for maintaining dynamic set partitions, where his principal motivation is VLSI design [27]. The time bounds for this data structure are based on an involved accounting argument. Calinescu's data structure supports the operations Insert, Delete, and Report in $O(n)$ amortized time, and thus is faster than all data structures described in this paper except for the Monte Carlo data structure (Sections 3.1 and 3.3) and the geometric-set-partitions data structure (Section 5).

Calinescu's data structure and our Monte Carlo data structure each have several advantages. Calinescu's data structure is deterministic, and consequently has no probability of error; in contrast, our Monte Carlo data structure has a polynomially small error probability. On the other hand, our Monte Carlo data structure runs in $O(n)$ *worst-case* time, whereas Calinescu's time bounds are amortized. Furthermore, Calinescu's results holds only for by bipartitions, whereas our Monte Carlo data structure applies to general set partitions. Finally, Calinescu's data structure is complicated, whereas our Monte Carlo data structure is simple to understand and implement. As a result, we believe that our Monte Carlo data structure is the natural data structure to code for the applications of decision-tree construction and VLSI design [8].

2 Basic Results: Deterministically Maintaining Set Partitions

In this section, we present an efficient data structure for deterministically maintaining set partitions under the operations *insert*, *delete*, and *report*. For *delete*, we assume that we are given a pointer to the set partition in question and hence defer the issue of retrieving these pointers to an auxiliary dictionary data structure.

Notation. We use the following notation throughout the paper. We let the universal set $U = \{1, \ldots, n\}$. Each set partition P partitions U into parts(P) disjoint subsets $P^{(1)}, \ldots, P^{(\mathrm{parts}(P))}$ such that $\cup_{i=1}^{\mathrm{parts}(P)} P^{(i)} = U$. We let part$(P, i)$ denote the part of P containing element i. Thus, n represents the size of the universe U and is fixed, and k represents the number of partitions and changes over time.

We say that a parameterized event E_p occurs *with high probability* if for any constant $c > 0$ there exists a valid choice of parameter p such that $\Pr[E_p] \geq 1 - n^{-c}$.

Lemma 1 *Let A and B be set partitions of $U = \{1, \ldots, n\}$. The induced partition (or refinement) of A and B can be computed in $O(n)$ time.*

Proof: Each element $i \in U$ can be represented by the three-character string $(\mathrm{part}(A, i), \mathrm{part}(B, i), i)$. All pairs of strings whose first two elements are identical represent elements that are in the same part in the refined partition of A, B. These strings can be radix sorted in linear time, after which all elements will be grouped by refined part. A linear sweep can now associate each element with its part in the refined partition. ∎

Repeated application of Lemma 1 yields a data structure that supports insert and report operations in linear time but does not explicitly support deletions. A naïve solution could recompute the induced partition from scratch on each deletion by repeatedly applying Lemma 1, for a total cost of $O(kn)$ per deletion.

Lemma 2 *Set partitions can be dynamically maintained such that the insertion and deletion operations take $O(n \log k)$ time, while reports can be performed in $O(n)$ time.*

Proof: We maintain a balanced binary tree T whose k leaves comprise the set of input partitions S, and each intermediate node is the induced partition of its two children. Therefore, the root of T represents the induced partition of S, and can be produced in linear time to satisfy a report query.

Insertion and deletion can be implemented as in any balanced binary tree such as [20]. Insertions and deletions in a red-black tree require $O(1)$ rotations in the worst case. Although each rotation affects only a constant number of nodes, the induced partitions on all $O(\log k)$ intermediate root-to-leaf nodes must be recomputed using Lemma 1, so that the time required for insertion and/or deletion is $O(n \log k)$. ∎

We note that a similar structure, called a partition tree, appears in a different context in Yellin [40]. We can modify the data structure of Lemma 2 to reduce the complexity of insertion to linear; this enhancement is effective when there are substantially more insertions than deletions.

Theorem 1 *Set partitions can be maintained with $O(n)$ time insertion and report operations, and $O(n \log k)$ time deletions. The space usage is $O(kn)$ machine words or $O(kn \log n)$ bits.*

Proof: In addition to a balanced binary tree T we maintain a separate global induced partition G. Initially G is a partition containing just one set. Partition G is always the induced partition of the k current partitions. Thus, for a report we return G, which takes $O(n)$ time.

We now describe an amortized solution, which we then deamortize. At every insertion we update G in $O(n)$ time. However, instead of updating T after every insertion, we wait until we have accumulated $\log k$ insertions. If a deletion occurs before $\log k$ insertions have accumulated, then we create an auxiliary balanced binary tree T' with the less than $\log k$ accumulated partitions, which takes $O(n \log k)$ time because we have to create $O(\log k)$ intermediate nodes in this tree each taking $O(n)$ time by Lemma 1. Then we merge T' with T in $O(n \log k)$ time. This is possible because two balanced binary trees of sizes s_1 and s_2 can be merged in $O(\log(s_1 + s_2))$ time making only $O(1)$ modifications [26]. Although only a constant number of nodes are affected, the induced partitions on all $O(\log k)$ intermediate root-to-leaf nodes must be recomputed using Lemma 1, so the time required for merging is $O(n \log k)$. We charge this time to the following deletion.

If we accumulate $\log k$ insertions, then we create an auxiliary balanced binary tree T' of size $\log k$ in $O(n \log k)$ time and merge T' with T in $O(n \log k)$ time. Since we spend $O(n \log k)$ time for every $\log k$ insertions, this scheme performs insertions in amortized $O(n)$ time.

We remove the amortization by distributing the $O(n \log k)$ work for tree building and merging over the next $\log k$ insertions. If a deletion is required before the next $\log k$ insertions, we perform the remaining work for building and merging before the deletion and charge it to the deletion. Specifically, we finish any leftover work from up to $2 \log k$ previous insertions as explained above, which requires $O(n \log k)$ time. Then we perform a deletion as in the proof of Lemma 2, which also takes $O(n \log k)$ time. Finally we update G with the root of T in $O(n)$ time. Thus deletions run in $O(n \log k)$ time.

This data structure maintains $O(k)$ partitions. The space used is $O(n)$ machine words per partition for a total of $O(kn)$ machine words or $O(kn \log n)$ bits. ∎

3 Randomized Data Structures for Maintaining Set Partitions

In this section we present randomized data structures for maintaining set partitions. First we assume that each inserted partition is a *bipartition*, which means that the elements separate into exactly two subsets. We present both Monte Carlo and Las Vegas algorithms for this case. These results easily generalize to D-partitions (partitions breaking the elements into D sets) but where the cost increases by a factor of $\log D$. Finally, we show how to modify the Monte Carlo algorithm to run with general partitions at no extra asymptotic cost.

3.1 Monte Carlo Data Structure

Colors of Elements. We first describe how to maintain the partition information. At each step t of the algorithm an integer $C_t[i]$ is associated with each element i; we call integer $C_t[i]$ the *color* of i. Specifically, $C_t[i] \in \{0, \ldots, R-1\}$, where R has size polynomial in n, that is, $R \in O(n^c)$ for some constant c. Thus, $C_t[i]$ is represented using $O(\log n)$ bits, which fit within $O(1)$ machine words. We maintain the following invariant:

Invariant 3 *With high probability, two elements i and j are in the same set at step t in the induced partition if and only if they have the same color, that is, $C_t[i] = C_t[j]$.*

We store (a weighted version of) the partitions $P_1 \ldots P_k$ that comprise S in a simple structure that we call a *partition list*, where the partitions are ordered by increasing insertion time. We insert or delete any partition in the partition list in time $O(n + \log k) \in O(n)$ using a balanced tree or some other basic data structure. This insertion/deletion cost follows because there are at most 2^n different bipartitions, and hence $k \leq 2^n$.

Insert. The k-th bipartition (last one inserted) is supplied as a 0-1 array $P_k[1 \ldots n]$, where $P_k[i] \in \{0, 1\}$. We insert this bipartition in step t as follows:

> I-1 $r_k :=$ randomly chosen integer $\in \{1 \ldots R-1\}$ for the k-th bipartition.
>
> I-2 $P_k[i] := r_k \cdot P_k[i]$, for $i = 1 \ldots n$. ($P_k[i]$ is now the k-th bipartition weighted by the randomly chosen integer r_k.)
>
> I-3 $C_t[i] := (C_{t-1}[i] + P_k[i]) \bmod R$, for $i = 1 \ldots n$.
>
> I-4 Store $P_k[1 \ldots n]$ in the list of (weighted) bipartitions.

Once we have calculated color $C_t[i]$, we no longer need to store $C_{t-1}[i]$.

Delete. We delete a bipartition P_u $(1 \leq u \leq k)$ in step t as follows.

> D-1 Find $P_u[1 \ldots n]$ in the partition list.
>
> D-2 $C_t[i] := (C_{t-1}[i] - P_u[i]) \bmod R$, for $i = 1 \ldots n$.
>
> D-3 Delete $P_u[1 \ldots n]$ from the partition list.

Report. To report, we radix sort the elements by color and report the partition of elements induced by their colors. Since colors are $O(\log n)$-bit integers, radix sort takes $O(n)$ time [11].

The Monte Carlo data structure has the following complexity:

Theorem 2 *In the Monte Carlo data structure, insert, delete, and report operations run in time $O(n)$. The space usage is $O(kn + n\log n)$ bits or $O(kn/\log n + n)$ machine words.*

Proof: Since machine words have $\Omega(\log n)$ bits, an integer of size at most R is represented in $O(1)$ machine words. In an insertion, Step I-1 requires $O(\log R) \in O(\log n)$ random bits. Steps I-2 and I-3 each requires time $O(n)$. Step I-4 requires time $O(n + \log k) = O(n)$ to store the partition.

In a deletion, Steps D-1 and D-3 each require $O(n + \log k) = O(n)$ time to access the partition list. Step D-2 requires $O(n)$ time to scan the partition.

The space used to store the colors is $O(n)$ machine words or $O(n \log n)$ bits, and the space used to store the k partitions is $O(n)$ bits per partition for a total of $O(kn)$ bits or $O(kn/\log n)$ machine words. ∎

Probability of Error. Errors are one-sided. If two elements have different colors, they always belong to different sets of S. An error occurs whenever two elements are assigned the same color but actually belong to different sets.

Theorem 3 *In the Monte Carlo data structure if the algorithm runs for a polynomial number of steps, then for sufficiently large $R = O(n^c)$ all report operations run correctly with high probability.*

Proof: We say that step t is a *bad step* if an error is introduced in this step. More specifically, we say that step t is a *bad step for elements i and j* if i and j belong to different sets and have different colors in step $t-1$, i.e.,

$$C_{t-1}[i] \neq C_{t-1}[j],$$

but erroneously have the same color in step t, i.e.,

$$C_t[i] = C_t[j].$$

In each step t, a partition $P_u[1 \dots n]$ is either inserted or deleted. Assume without loss of generality that $P_u[i] = 0$ and that $P_u[j] = r_u$. On an insertion, there is a bad step for i and j iff

$$r_u = C_{t-1}[i] - C_{t-1}[j] \mod R.$$

On a deletion, there is a bad step for i and j iff

$$r_u = C_{t-1}[j] - C_{t-1}[i] \mod R.$$

In either case, the probability that an error is introduced for i and j is $1/(R-1)$. Thus the probability that t is a bad step for *any* i and j is at most $n^2/(R-1)$. Finally, the probability that there is an error in any step until step t is bounded by $tn^2/(R-1)$. The theorem follows for a sufficiently large R. ∎

Because the probability of an error is polynomially small, the data structure should run correctly for a polynomial number of steps determined by the size of R. Practical considerations may dictate that $R < 2^{64}$ so that all computations fit within one 64-bit machine word. In this case, one can obtain smaller error probabilities by running multiple versions of the data structure, where each version has independently chosen colors. Alternatively, perhaps one could save randomness by using multiple values of R and the same coin flips.

We anticipate that the Monte Carlo data structure will be extremely fast because it only uses a small number of additions and subtractions. It is interesting to note that our Monte Carlo data structure suffices for the practical application of building a tree that distinguishes all objects using a small number of probes. Our randomized scheme will never classify two objects as different which are in fact indistinguishable, and hence the only consequence of being unlucky is to add a small number of additional probes to the test set, that is, to increase the height of the decision tree by a small amount. Since absolutely minimizing the number of probes in a test set, i.e., minimizing the height of a decision tree, is NP-complete [5, 17, 18], this penalty is excusable.

We conclude this section by explaining why these results generalize to D-partitions (partitions breaking the elements into D sets) with a $\log D$-factor cost increase. Our construction is as follows: We number the D sets from $1 \ldots D$ and represent the sets using $\log D$ bits. Then, we replace each D-partition by $\log D$ bipartitions that induce the original D-partition, where there is one bipartition for each bit position. Let $P[1 \ldots n]$ be the partition associated with the b-th bit. Then $P[i] = 0$ if i is in a set where the b-th bit of the set number is 0, and otherwise $P[i] = 1$.

3.2 Las Vegas Data Structure

The time complexity for the Las Vegas data structure is only marginally slower than for the Monte Carlo data structure, although now it is amortized. In order to make the data structure Las Vegas, we remove the probability of error from Invariant 3. Now each time we perform an insertion or deletion, we *verify* that this invariant holds.

Verifying an Insertion. We verify an insertion of permutation P_k in time step t by including a verification phase at the end of the operation:

I-5 Verify that for all i, j, $(C_t[i] = C_t[j]) \Longrightarrow (C_{t-1}[i] = C_{t-1}[j])$.

I-6 If so, continue. If not, an *error* is found. Spawn off an independent execution or run any other alternative protocol.

Step I-5 is executed in linear time by maintaining additional structure of the elements. Namely, in each step we store the elements in sorted order by increasing color, and elements with the same color are ordered by increasing element number. We store these elements in an array $\Pi_t[1 \ldots n]$, where $\Pi_t[\ell] = j$ means that in step t, element j has the ℓ-th position in the sorted order.

Observe that if the ordering $\Pi_{t-1}[1 \ldots n]$ in step $t-1$ is known, the ordering $\Pi_t[1 \ldots n]$ can be computed in linear time by merging three sorted (and interleaved) lists:

- the elements of $\Pi_t[1 \ldots n]$ whose colors do not change between step $t-1$ and step t,
- the elements of $\Pi_t[1 \ldots n]$ whose colors increase by r_k, and
- the elements of $\Pi_t[1 \ldots n]$ whose colors increase by r_k and then (by the rules of modular arithmetic) decrease by R.

Thus, Step I-5 is divided into two substeps:

I-5a Compute $\Pi_t[1 \ldots n]$ from $\Pi_{t-1}[1 \ldots n]$ by merging three lists.

I-5b Verify that for $\ell = 1 \ldots n-1$,

$$\left(C_t\Big[\Pi_t[\ell]\Big] = C_t\Big[\Pi_t[\ell+1]\Big]\right) \Longrightarrow \left(C_{t-1}\Big[\Pi_t[\ell]\Big] = C_{t-1}\Big[\Pi_t[\ell+1]\Big]\right).$$

The insertion verification algorithm has the following performance:

Lemma 4 *In the Las Vegas data structure, verification of insertions runs in time $O(n)$. There is no asymptotic increase in space usage when performing verification of insertions; the space usage when storing k bipartitions is $O(kn + n\log n)$ bits or $O(kn/\log n + n)$ machine words.*

Proof: Step I-5a requires linear time because it entails merging three lists. Step I-5b requires linear time because we only compare the color of each element $\Pi_t[\ell]$ with its neighboring element $\Pi_t[\ell+1]$ in the ordering. The analysis for space usage appears in the proof of Theorem 2. ∎

Verifying a Deletion. We now show how to verify the deletion of partition P_u ($1 \leq u \leq k$) in step t. Let $P_1 \ldots P_k$ be the existing partitions in S just before P_u is deleted, where the partitions are ordered by insertion time.

Verifying deletions is more complicated for the following reasons. Suppose that after the deletion of a partition $P_u[1 \ldots n]$, two elements i and j have the same color. We do not know *a priori* whether this is because the last partition, P_u, separating i and j has been removed, or whether i and j are erroneously assigned the same color and are in fact separated by other partitions. We verify deletions as follows.

D-4 Verify that if $C_t[i] = C_t[j]$, then i and j are in the same set in all bipartitions $P_1, \ldots, P_{u-1}, P_{u+1}, \ldots, P_k$.

D-5 If so, continue. If not, an *error* is found. Spawn off an independent execution or run any other alternative protocol.

Note that Step D-4 could potentially be expensive because it may involve scanning through the entire list of partitions $P_1, \ldots, P_{u-1}, P_{u+1}, \ldots, P_k$. Despite this expense, we show that the amortized cost of verifying insertions and deletions is $O(n\alpha(n))$, where $\alpha(n)$ is the inverse Ackerman function. To bound this cost, we show that for verification, each partition is examined $O(n)$ times and each examination requires amortized time $O(\alpha(n))$. As with insertions, we maintain the elements in sorted order and this again involves merging three sorted lists. (The merge is minimally different; we now add instead of subtracting and subtract instead of adding.)

We focus on each prefix $P_1, \ldots, P_m$ $(1 \leq m \leq k)$ of partitions $P_1, \ldots, P_k$. Let $C^{(m)}[i]$ denote the color of element i after inserting partitions $P_1 \ldots P_m$, for $i = 1 \ldots n$. Note that $C^{(m)}[i]$ is just notation; it is not explicitly stored by the data structure. Observe that $C^{(m)}[i]$ may change over time if earlier partitions are deleted.

We maintain a union find data structure $\textsc{Union-Find}^{(m)}$ for each prefix $P_1, \ldots, P_m$. The data structure $\textsc{Union-Find}^{(m)}$ is initialized when the partition P_m is first inserted, and this insertion uses $O(n)$ time. Thus, two elements i and j belong to the same set in the data structure $\textsc{Union-Find}^{(m)}$ if they belong to the same set in all of the partitions $P_1, \ldots, P_m$. Because partitions are always inserted *at the end* of S, the sets in each $\textsc{Union-Find}^{(m)}$ data structure may coalesce when earlier partitions are deleted but never split. The sets in a union find data structure can combine together at most $n - 1$ times before a single set remains and consequently all elements have the same color. The operation $\textsc{Find-Set}^{(m)}[i]$ locates the *smallest* element (according to the original total order of U) belonging to the same set as element i in the $\textsc{Union-Find}^{(m)}$ data structure. The operation $\textsc{Union}^{(m)}[i, j]$ *combines* the set containing i and the set containing j in the $\textsc{Union-Find}^{(m)}$ data structure.

We divide Step D-4 into substeps as follows:

D-4a Compute $\Pi_t[1 \ldots n]$ from $\Pi_{t-1}[1 \ldots n]$ by merging three lists.

D-4b By incrementing ℓ, find all values of ℓ such that

$$C_t[\Pi_t[\ell]] = C_t[\Pi_t[\ell+1]]\,,$$

but

$$\textsc{Find-Set}^{(k)}[\Pi_t[\ell]] \neq \textsc{Find-Set}^{(k)}[\Pi_t[\ell+1]]\,.$$

This means that $C_{t-1}[\Pi_t[\ell]] \neq C_{t-1}[\Pi_t[\ell+1]]$. (This step identifies pairs of elements that have the same color currently but had different colors in the previous step.)

D-4c For all such ℓ, execute the following:

- Verify that
$$C^{(u-1)}[\Pi_t[\ell]] = C^{(u-1)}[\Pi_t[\ell+1]]$$
by checking that
$$\textsc{Find-Set}^{(u-1)}[\Pi_t[\ell]] = \textsc{Find-Set}^{(u-1)}[\Pi_t[\ell+1]]\,.$$
If this check fails, then an *error* is found. (This step verifies that these elements are in the same set in the induced set partition of $P_1 \ldots P_{u-1}$.)
- Verify that for all partitions $P_m = P_{u+1} \ldots P_k$,
$$P_m\,[\Pi_t[\ell]] = P_m\,[\Pi_t[\ell+1]]\ .$$
If this check fails, then an *error* is found. (This step verifies that these elements have the same color in the bipartitions $P_{u+1} \ldots P_k$.)
- For all bipartitions $P_m = P_{u+1} \ldots P_k$,
$$\textsc{Union}^{(m)}[\Pi_t[\ell]\,,\,\Pi_t[\ell+1]].$$

The deletion verification algorithm has the following performance:

Lemma 5 *In the Las Vegas data structure, a sequence of τ insertions and deletions starting with a null S takes time $O(\tau n\alpha(n))$. The space when storing k bipartitions is $O(kn + n\log n)$ bits or $O(kn/\log n + n)$ machine words.*

Proof: Let $\tau = \tau_i + \tau_d$, where τ_i is the number of insertions and τ_d is the number of deletions, implying that the number of partitions currently stored is $k = \tau_i - \tau_d$. Step D-4a requires $O(n)$ time because it entails merging three lists. Step D-4b requires $O(n)$ time because each check takes constant time and there are $n-1$ checks. Therefore for τ_d deletions, these two steps take total time $O(\tau_d n)$.

During the execution of τ insertions and deletions, the time spent in Step D-4c is $O(n\alpha(n))$ per partition because there are at most $n-1$ set unions per partition. Since there can be at most τ_i

partitions, the total time used by Step D-4c is $O(\tau_i n\alpha(n))$; the cost for the second bullet of the step is dominated by the costs for the third bullet of the step. Hence the total time spent in deletion verification for any sequence of τ insertions and deletions is $O(\tau_i n\alpha(n) + \tau_d n)$, which is $O(\tau n\alpha(n))$.

The analysis for space usage appears in the proof of Theorem 2. ∎

If a verification identifies an error, then we run an alternative protocol. Because the probability of an error is polynomially small, we obtain the following performance bounds for the Las Vegas data structure.

Theorem 4 *In the Las Vegas data structure report operations run in time $O(n)$ in expectation and with high probability, and any sequence of τ insert and delete operations starting with null S runs in time $O(\tau n\alpha(n))$ in expectation and with high probability.*

3.3 Monte Carlo Data Structure for General Partitions

While most of the data structures in this paper are designed for bipartitions, the Monte Carlo data structure (Section 3.1) can be generalized so that it runs with any set partition. The resulting time complexity stays the same. Recall that set partition P_k divides U into parts(P_k) disjoint subsets $P_k^{(1)}, \ldots, P_k^{(\mathrm{parts}(P))}$ such that $\cup_{i=1}^{\mathrm{parts}(P_k)} P_k^{(i)} = U$.

On insertions, we associate a random number $r_{jk} \in \{0, \ldots, R-1\}$ with each part $P_k^{(j)}$ of the k-th partition, for $j = 2, \ldots, \mathrm{parts}(P_k)$, and we let $r_{1k} = 0$. Then we encode the k-th partition as an array so that

$$P[i] = r_{jk} \text{ iff } i \in P_k^{(j)}.$$

The rest of the insertion proceeds as in Section 3.1. In our presentation we assume that a random number can be obtained in $O(1)$ time. At the end of this section we show how to modify the data structure to use fewer random bits. Thus, the insertion procedure is as follows:

I-1 $r_{jk} := \begin{cases} \text{randomly chosen integer } \in \{0, \ldots, R-1\}, & \text{for } j = 2 \ldots \mathrm{parts}(P_k), \\ 0, & \text{for } j = 1. \end{cases}$

I-2 $P_k[i] := r_{jk}$, iff $i \in P_k^{(j)}$.

I-3 $C_t[i] := (C_{t-1}[i] + P_k[i]) \bmod R$, for $i = 1 \ldots n$.

I-4 Store $P_k[1 \ldots n]$ in the list of (weighted) partitions.

The deletion procedure is unchanged from Section 3.1.

The Monte Carlo data structure for general set partitions has the following complexity:

Theorem 5 *In the Monte Carlo data structure for general set partitions, insert, delete, and report operations run in time $O(n)$. The space usage is $O(kn\log n)$ bits or $O(kn)$ machine words.*

Proof: Step I-1 runs in $O(n)$ time, because of our temporary assumption that random numbers are obtained in constant time. Steps I-2 and I-3 each requires time $O(n)$. Step I-4 requires time

$O(n + \log k) = O(n)$ to store the partition because the data structure is only guaranteed to run correctly for a polynomial amount of time and therefore k is polynomial in n. The analysis for deletions and space usage is similar to the proof of Theorem 2 except that that it requires $O(n \log n)$ bits to store a general partition. ∎

As with the Monte Carlo data structure for bipartitions, errors are one-sided. An error occurs whenever two elements are assigned the same color but actually belong to different sets.

Theorem 6 *In the Monte Carlo data structure for general set partitions, if the algorithm runs for a polynomial number of steps, then for sufficiently large $R = O(n^c)$ all report operations run correctly with high probability.*

Proof: We say that step t is a *bad step for elements i and j* if an error is introduced in this step. There are two types of errors. First, an error can be introduced upon insertion of partition P_k if two different sets of the set partition are erroneously assigned the same random number, that is, $r_{ik} = r_{jk}$, for $i \neq j$, $1 \leq i, j \leq \text{parts}(P_k)$. The probability that this type of error is introduced in step t is at most n^2/R. Thus, the probability that this error is introduced at any time before step t is at most tn^2/R.

As in the Monte Carlo data structure for bipartitions, step t is also a bad step for elements i and j if i and j belong to different sets and have different colors in step $t-1$, that is, $C_{t-1}[i] \neq C_{t-1}[j]$, but erroneously have the same color in step t, that is, $C_t[i] = C_t[j]$. In each step t, a partition $P_u[1 \ldots n]$ is either inserted or deleted. Assume that $P_u[i] = r_{iu}$ and that $P_u[j] = r_{ju}$. On an insertion, this type of error occurs for i and j iff

$$r_{ju} - r_{iu} = C_{t-1}[i] - C_{t-1}[j] \mod R.$$

On a deletion, this type of error occurs for i and j iff

$$r_{ju} - r_{iu} = C_{t-1}[j] - C_{t-1}[i] \mod R.$$

In either case, the probability that this error is introduced for i and j is $1/R$. Thus the probability that this type of error is introduced in step t for *any* i and j is bounded by n^2/R. Finally, the probability that this type of error occurs in any step until step t is bounded by tn^2/R. Hence, the probability that either type of error occurs in any step until step t is bounded by $2tn^2/R$. The theorem follows for a sufficiently large R. ∎

Even if the generation of random numbers is expensive, we retain the optimal running time by saving randomness: the proof of Theorem 6 only relies on pairwise independence of the r_{jk} variables. Therefore, instead of choosing a random number for each piece of a set partition, we use a standard 2-universal hash function [30]. Specifically, we choose a value for $r_{jk} \in \{0, \ldots, R-1\}$ as follows: We pick a unique ID for partition P_k (e.g., the time it was inserted). Then we hash based on j and this unique ID, and we set r_{jk} to this hash value; this hashing takes constant worst-case time.

4 Estimating the Number of Partitions Separating Elements

When building a decision tree (e.g., for OCR), one may want to maintain more detailed information besides the induced partition. For fault tolerance, one may insist that each element be separated by at least Δ partitions. Thus, for each pair of elements the data structure could store and return the *number* of partitions that separate the elements.

Because the data structure queries supply more information, one can reasonably suppose that the operations run more slowly. This is certainly true in a naïve solution to this problem, where the data structure maintains a two-dimensional array

Count$[1 \ldots n][1 \ldots n]$, where Count$[i][j] = d$ signifies that d partitions separate the elements i and j. Thus, the operations *insert* and *delete* increment or decrement the appropriate array values. Unfortunately, a single insert or delete can require $O(n^2)$ modifications, which may be prohibitively expensive for large n.

On the other hand, it may not be necessary to know the *exact* number of partitions that separate elements i and j. In applications such as building decision trees, *approximate* knowledge of this number may be satisfactory. This is the problem we explore in this section for bipartitions.

Our data structure supports the following three operations.

- *Insert*(P, S) – add a new bipartition P to the set of bipartitions S.
- *Delete*(P, S) – delete the existing bipartition P from the set of bipartitions S.
- *Query*(i, j, S) – output an estimate of the number of bipartitions in S separating elements i and j.

We show that the operations *insert* and *delete* can be implemented to run in worst-case time $O(n \log n)$. With some word-level optimizations *insert* and *delete* can be implemented to run in amortized time $O(n \log \log n)$. *Query* requires logarithmic time and returns an answer that is accurate to within any prespecified constant factor.

As in Section 3, we assign integers called *colors* to elements. Each element i has $\beta \log n$ separate colors, C$[i][1 \ldots \beta \log n]$, initialized to zero.

4.1 Insert and Delete

A new bipartition is supplied as an array P$[1 \ldots n]$, where P$[i] \in \{0, 1\}$. Note that each bipartition has two representations, where one representation is the complement of the other. To insert a bipartition, independently modify each of the $\beta \log n$ colors of the elements as follows:

- Randomly choose one of the two representations of the bipartitions for each of the $\beta \log n$ colors.
- Add these values to the colors of the elements.

Thus, for each element some of the colors are incremented by 1 and some of the colors remain the same.

To delete a bipartition, reverse the changes made to each of the $\beta \log n$ colors when the bipartition was inserted. Thus, each composite color either remains unmodified or decreases by 1.

4.2 Queries

To estimate the number of bipartitions separating elements i and j, we compare the $\beta \log n$ colors of i and j. Our estimation is based on the following variables:

$$\delta_\ell[i][j] = \Big| \mathrm{C}[i][\ell] - \mathrm{C}[j][\ell] \Big|, \quad (1 \le \ell \le \beta \log n).$$

When there is no ambiguity we abbreviate $\delta_\ell[i][j]$ by δ_ℓ. By linearity of expectation, the expected values of all the colors of the elements are identical and are half the current number of partitions in the partition list:

Lemma 6 *The expected value of each color is $k/2$. That is, for all $1 \leq i \leq n$*

$$E(C[i][1]) = E(C[i][2]) = \cdots = E(C[i][\beta \log n]) = k/2.$$

Naturally, the actual colors may deviate from this expected value, and we use this deviation in our estimation procedure. If $d = 0$, i.e., no bipartitions separate i and j, then C$[i][\ell]$ =C$[j][\ell]$, for all $\ell = 1 \ldots \beta \log n$. The less the values of C$[i][\ell]$ and C$[j][\ell]$ are correlated, the more bipartitions separate i and j. We can view the distribution on $\delta_\ell = |\mathrm{C}[i][\ell] - \mathrm{C}[j][\ell]|$ as a random walk, as explained in the following lemma:

Lemma 7 *Suppose that d bipartitions separate elements i and j. Consider a one-dimensional random walk of length d that starts at the origin and at each step moves one unit to the right with probability $1/2$ and one unit to the left with probability $1/2$. The probability that $\delta_\ell = z$ equals the probability that the random walker ends at a distance z from the origin.*

Proof: The bipartitions that do not separate elements i and j contribute nothing to $\delta_\ell = |\mathrm{C}[i][\ell] - \mathrm{C}[j][\ell]|$. The bipartitions that do separate i and j either add or subtract one from δ_ℓ depending on which representation is used. Since there are d such bipartitions and their representations are chosen with equal probability for each ℓ, the lemma follows. ∎

Using Lemma 7, the probability $\Pr[\delta_\ell = z]$ can be calculated directly for small d and can be approximated for large d using the following theorem from Feller [15]:

Theorem 7 ([15], Page 76) *Let S_d be the (positive or negative) position of a random walker on the integer line after d steps. Then as $d \to \infty$,*

$$\Pr\left[S_d > x\sqrt{d}\right] \to 1 - \Re(x),$$

where $\Re$ stands for the normal distribution function defined as

$$\Re(x) = \frac{1}{\sqrt{2\pi}} \int_{\zeta=-\infty}^{x} e^{-\zeta^2/2}\, d\zeta.$$

The following is a corollary to Theorem 7.

Corollary 8 *Let $|S_d|$ be the absolute distance from origin of a random walker on the integer line after d steps. Then as $d \to \infty$,*

$$\Pr\left[|S_d| \leq x\sqrt{d}\right] \to 2\Re(x) - 1,$$

and

$$\Pr\left[|S_d| > x\sqrt{d}\right] \to 2(1 - \Re(x)).$$

We introduce a constant γ that appears in our estimation algorithm.

Definition 9 *Let γ be a constant such that as $d \to \infty$,*

$$\Pr\left[|S_d| \le \gamma\sqrt{d}\right] \to \frac{1}{2}.$$

The definition of γ and Corollary 8 imply that $\Re(\gamma) = 0.75$ and hence $\gamma \approx 0.6745$.

4.3 Estimating Large d

We first show how to estimate d when d is sufficiently large. We only require d to be greater than a sufficiently large constant. This constant is independent of n but is a function of error parameter ε and constant c in the definition of high probability.

Let $\hat{d}$ be our estimate of d. For large enough d, we evaluate $\hat{d}$ as follows:

E-1 Compute the median δ_{med} of $\delta_1 \ldots \delta_{\beta \log n}$.

E-2 Let $\hat{d} = \left(\frac{\delta_{\text{med}}}{\gamma}\right)^2$.

We prove our error bounds by using the following statement of Chernoff bounds from [30].

Theorem 8 ([30]) *Let $X_1, \ldots, X_m$ be independent Poisson trials such that, for $1 \le i \le m$, $\Pr[X_i = 1] = p_i$, where, $0 < p_i < 1$. Then, for the random variable $X = \sum_{i=1}^m X_i$, $\mu = E[X] = \sum_{i=1}^m p_i$, and any $\delta > 0$,*

$$\Pr[X > (1+\delta)\mu] < \left[\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right]^\mu,$$

and for all $0 < \delta \le 1$,

$$\Pr[X < (1-\delta)\mu] < e^{-\mu\delta^2/2}.$$

We now give the error bounds for this estimator:

Theorem 9 *For any constant (independent of n and d) error parameter ε and constant c, there is a constant β such that the following holds with probability at least $1 - 1/n^c$ (with high probability): If the number of bipartitions separating elements i and j is d, then the estimate $\hat{d}$ of d is bounded by*

$$(1-\varepsilon)d < \hat{d} \le (1+\varepsilon)d.$$

Proof: We first provide a bound on the probability that $\hat{d} \le (1-\varepsilon)d$. Let $X_1 \ldots X_{\beta \log n}$ be 0/1-random variables where

$$X_\ell = \begin{cases} 1, & \text{if } \left(\frac{\delta_\ell}{\gamma}\right)^2 \le (1-\varepsilon)d, \\ 0, & \text{otherwise.} \end{cases}$$

Hence,

$$\begin{aligned}
\Pr[X_\ell = 1] &= \Pr\left[\left(\tfrac{\delta_\ell}{\gamma}\right)^2 \le (1-\varepsilon)d\right] \\
&= \Pr\left[\delta_\ell \le \gamma\sqrt{(1-\varepsilon)d}\right] \\
&= (1-\lambda)^2 \Pr\left[\delta_\ell \le \gamma\sqrt{d}\right],
\end{aligned}$$

where λ is a positive constant strictly less than 1 that is independent of n and d but is a function of ε. Using the definition of γ we have that as $d \to \infty$,

$$\Pr[X_\ell = 1] \to (1-\lambda)^2 \frac{1}{2}.$$

Hence for sufficiently large d independent of n,

$$\Pr[X_\ell = 1] < (1-\lambda)\frac{1}{2}.$$

Define $X = X_1 + X_2 + \cdots + X_{\beta \log n}$. By Chernoff bounds [30] (Theorem 8) for any ε and c, there exists a β such that with high probability $X < (\beta \log n)/2$. Therefore with high probability,

$$\hat{d} = \left(\frac{\delta_{\mathrm{med}}}{\gamma}\right)^2 > (1-\varepsilon)d$$

We now bound the probability that $\hat{d} > (1+\varepsilon)d$. Let $X'_1 \ldots X'_{\beta \log n}$ be 0/1-random variables where

$$X'_\ell = \begin{cases} 1, & \text{if } \left(\frac{\delta_\ell}{\gamma}\right)^2 > (1+\varepsilon)d, \\ 0, & \text{otherwise.} \end{cases}$$

Hence,

$$\begin{aligned}
\Pr[X'_\ell = 1] &= \Pr\left[\left(\tfrac{\delta_\ell}{\gamma}\right)^2 > (1+\varepsilon)d\right] \\
&= \Pr\left[\delta_\ell > \gamma\sqrt{(1+\varepsilon)d}\right] \\
&= (1-\lambda')^2 \Pr\left[\delta_\ell > \gamma\sqrt{d}\right],
\end{aligned}$$

where λ' is a positive constant strictly less than 1 that is independent of n and d but is a function of ε. Using the definition of γ we have that as $d \to \infty$,

$$\Pr[X'_\ell = 1] \to (1-\lambda')^2 \frac{1}{2}.$$

Hence for sufficiently large d independent of n,

$$\Pr[X'_\ell = 1] < (1-\lambda')\frac{1}{2}.$$

Define $X' = X'_1 + X'_2 + \cdots + X'_{\beta \log n}$. By Chernoff bounds [30] for any ε and c, there exists a β such that with high probability $X' < (\beta \log n)/2$. Therefore with high probability,

$$\hat{d} = \left(\frac{\delta_{\mathrm{med}}}{\gamma}\right)^2 \le (1+\varepsilon)d.$$

Thus, we have $(1-\varepsilon)d < \hat{d} \le (1+\varepsilon)d$. ■

We conclude that for any value of ε and error probability n^{-c}, there is a sufficiently large value of d (independent of n), such that $\hat{d} = (\delta_{\mathrm{med}}/\gamma)^2$ is a good estimate of d.

4.4 Calculating Small d Exactly

As we stated earlier, there is a constant value d_{thres} of d below which the estimator from Section 4.3 does not give high-probability guarantees. For example, this estimator cannot distinguish between $d = 0$ and $d = 2$ with high probability. However, when $d < d_{\text{thres}}$ we can find the exact value of d with high probability as follows.

Let S_d be the (positive or negative) position of a random walker on the integer line after d steps. The probability that $S_d = z$ is given by ([15], Page 75):

$$\Pr[S_d = z] = \binom{d}{\frac{d+z}{2}} 2^{-n},$$

where the binomial coefficient is to be interpreted as zero unless $(d+z)/2$ is an integer between 0 and d, inclusive.

Note that for even d and odd z this probability is 0, and similarly for odd d and even z. Thus, all the δ_ℓ values are even for an even d and odd for an odd d. First suppose that the δ_ℓ values are even. Let random variable $F(d)$ be the fraction of the δ_ℓ values that are 0 for the $\beta \log n$ random walks each of d steps. Then the expected value of $F(d)$ is given by

$$E[F(d)] = \Pr[S_d = 0] = \binom{d}{d/2} 2^{-d}.$$

This expectation is distinct for each d. To calculate $\hat{d}$, we first determine the fraction $F(d)$ of the δ_ℓ values that are 0. By Chernoff bounds, there exists a sufficiently large constant β such that this fraction is within a $1 + \varepsilon$ multiplicative factor of its expectation with high probability. Hence, we let $\hat{d}$ be the integer between zero and $d_{\text{thres}} + 1$ that minimizes $|F(\hat{d}) - F(d)|$. We can choose ε to be sufficiently small so that $\hat{d}$ is exactly d with high probability. If we calculate $\hat{d}$ to be $d_{\text{thres}} + 1$, then the procedure for small d may be inaccurate and we need the estimation procedure for large d.

If δ_ℓ values are odd, we use a similar procedure that determines d by counting the number of δ_ℓ values that are 1.

4.5 Time Complexity

We present two implementations of our data structure for estimating the number of bipartitions separating elements. First we analyze a straightforward implementation, and then we give a more sophisticated, faster implementation.

The straightforward implementation is to store each color in its own machine word (or $O(1)$ machine words) because each color uses $O(\log n)$ bits and a word contains $\Omega(\log n)$ bits. This first strategy performs as follows:

Theorem 10 *The first implementation of the data structure for estimating the number of bipartitions separating elements has the following complexity: insertions and deletions run in $O(n \log n)$ time, and queries run in $O(\log n)$ time. The space usage is $O(kn + n \log^2 n)$ bits, or $O(kn/\log n + n \log n)$ machine words.*

Proof: In each insert or delete, we increment or decrement at most $n\beta \log n$ colors. Since the data structure is only guaranteed to work for a polynomial number of steps, and the colors change by at most 1 in each step, each color is only polynomially large and therefore is stored in $O(1)$ machine words. The time for an update is $O(n \log n)$.

To answer Query(i, j, S) we subtract the colors for i from the colors for j and take the absolute value. These subtractions give us the $\delta_1, \ldots, \delta_{\beta \log n}$ variables. We then find the median δ_{med}, which requires linear time in the number of values and hence takes $O(\log n)$ time. Therefore the time to answer a query is $O(\log n)$.

A bipartition requires n bits of storage, which fit in $O(n/\log n)$ machine words. Therefore the space usage required to store all k bipartitions is kn bits, which fit in $O(kn/\log n)$ machine words. Because there are $O(n)$ elements, each of which has $n\beta \log n$ $O(\log n)$-bit colors, the space usage for storing the colors is $O(n \log^2 n)$ bits or $O(n \log n)$ machine words. ∎

We now show that assigning each color to one (or $O(1)$) machine words is suboptimal. A better strategy is to store the $\beta \log n$ colors of each element as follows. Store the first bit of $\beta \log n$ colors for an element in a first word (or group of $O(1)$ words), the second bit of $\beta \log n$ colors for the element in a second word, etc. Thus, the bits of any one color are spread across $O(\log n)$ machine words. This allows faster insertions and deletions without increasing the cost of queries.

To make the above strategy work, we need an additional idea. We store each color in *two* counters such that the sum of the two counters equals the color. Now we store the counters according to the above strategy. On updates, we only *increment* the first counter and we only *decrement* the second. On queries, we add the counters together to reconstruct the color; then we store the color in the second counter, and set the first counter to 0. An alternative approach is to store and manipulate the colors in memory using a nonunique representation (see, e.g., [31]), and this obviates the need for two counters.

Our second implementation relies on the following standard data-structures subroutine.

Lemma 10 *The number of* 1*'s in a given subset of bits of a machine word can be counted in constant time.*

Proof: We first mask out the bits that are not in the given subset by changing them to zeros. We then count the number of 1's in the masked machine word. We implement this counting operation using a lookup table: we divide the word into segments of length $(\log n)/2$. The number of different bit strings of this size is only $\sqrt{n}$, and for each bit string we precompute the number of 1's. We therefore precompute the number of 1's in all possible strings and store the results in a table. Note that the table size and the precomputation time are both sublinear. This table allows us to count the number of 1's in constant time. ∎

The second implementation performs as follows:

Theorem 11 *The second implementation of the data structure for estimating the number of bipartitions separating elements has the following complexity: insertions and deletions run in amortized time* $O(n \log \log n)$*, and queries run in time* $O(\log n)$*. The space usage is* $O(kn + n \log^2 n)$ *bits, or* $O(kn/\log n + n \log n)$ *machine words.*

Proof: Because each color is stored in $O(\log n)$ machine words, we cannot use standard addition and subtraction of machine words to add colors in constant time. Instead, we effectively program a ripple-carry adder (see, for example, [31]), adding or subtracting $O(\log n)$ colors in parallel in $O(\log n)$ time.

Incrementing (respectively, decrementing) a counter involves adding (respectively, subtracting) 1 to the first bit of the counter. If the first bit is already 1, then a carry "ripples" to the second bit of the counter. If the second bit is already 1, then a carry "ripples" to the third bit of the counter, and so on. Notice that if there is a carry added to position b, then there are 2^b increments before another carry bit appears in position b.

We need to store the counters that we increment and the counters that we decrement *separately* in order to deliver the amortized constant bounds. This reveals the advantage of the nonunique representation: A counter stored with a nonunique representation can be incremented and decremented with a worst-case cost of $O(1)$ bit changes; see, e.g., [31] for details.

In our data structure we increment (respectively, decrement) $O(\log n)$ counters per element for each insert (respectively, delete). In the following we bound the time for updating the two sets of $\beta \log n$ counters for a single element.

We pay for accessing the b-th word (which stores the b-th bits of each counter) if even *one* of the $O(\log n)$ counters (partially) stored in the word is touched.

The b-th bit of each counter is modified at most a $1/2^b$ fraction of the time. In the worst case no bit modifications are done simultaneously. Hence, the b-th word is touched at most a $(\beta \log n)/2^b$ fraction of the time. Also, in the worst case we can touch the b-th word in every time step. Hence, the b-th word is accessed at most a $\min\left(\frac{\beta \log n}{2^b}, 1\right)$ fraction of the time. Therefore the amortized number of times we modify a word per update, summed over all bit positions, is at most

$$\sum_{b=1}^{O(\log n)} \min\left(\frac{\beta \log n}{2^b}, 1\right) = O(\log \log n).$$

We pay this cost for each element, hence the running time for insert/delete is $O(n \log \log n)$.

Observe that the carry operations in updating the counters are performed using the word-level parallelism of the RAM model: we only need standard masks and bit-wise logical operations, which belong to the instruction set of any computer.

We have now implemented fast updates, but because the colors are stored in a distributed manner, it is more involved to answer queries. As in Theorem 10, we subtract the colors for i and the colors for j and take the absolute value to obtain $\delta_1, \ldots, \delta_{\beta \log n}$ counters; we perform this subtraction using our ripple-carry adders.

To find the median of these $\beta \log n$ counters we use the following algorithm. For any subset Υ of the $\beta \log n$ counters each with ϕ bits, we recursively find the counter having rank ρ as follows: We count the number ρ_0 of 0's in the most-significant bits of the counters in Υ. Since the most-significant bits are all stored in $O(1)$ words, this operation can be performed in $O(1)$ time using Lemma 10.

If $\rho_0 \geq \rho$, then the most-significant bit of the counter having rank ρ is 0; we recursively find the counter having rank ρ among the subset of counters in Υ that have 0 as their most-significant bit. If $\rho_0 < \rho$, then the most-significant bit of the counter having rank ρ is 1; we recursively find the counter having rank $\rho - \rho_0$ among the subset of counters in Υ that have 1 as their most-significant bit.

For the recursive step we ignore the most-significant bits in this smaller subset of counters and treat the counters as having $\phi - 1$ bits. This step takes $O(1)$ time, and we have reduced the problem to a smaller problem in which each counter has one fewer bit. Since counters have $O(\log n)$ bits, this algorithm takes $O(\log n)$ time.

The bounds on space usage are proved as in Theorem 10. ∎

5 Maintaining Geometric Set Partitions

Suppose the elements in each set partition are points in the plane. We consider set partitions induced by a halfplane that distinguishes between the points that lie to the left or right of the defining line. Such partitions have been previously studied. For example, Freimer et al. [17, 18] prove that it is NP-complete to find the smallest subset of lines sufficient to shatter a point set, i.e., induce a complete partition of the points.

Clearly, the data-structure problem can be solved by testing all of the halfplanes against each point and reducing it to a non-geometric instance. However, exploiting the geometry simplifies the problem. The following operations are supported by such a data structure.

- *Insert-line*(L, S) – add a separating line L to the set of partitioning lines S.
- *Delete-line*(L, S) – delete the existing separating line L from the set of partitioning lines S.
- *Report*(S) – report the set partition of U induced by the set of partitioning lines S.

A naïve way to support these queries is by maintaining the arrangement of separating lines [12]. Any two points in the same cell of the arrangement represent unpartitioned elements. Report takes $O(n)$ time because we can maintain a list of non-empty cells in the arrangement with each cell maintaining a list of points in it. A line insertion/deletion into the arrangement takes $O(k+n)$ time, the former term for inserting/deleting a line in an arrangement of k lines [12] and the latter term to partition/merge the lists of points in the cells that get split/joined by the line.

This naïve method is faster than the data structures in the non-geometric setting when k is $O(n)$, but its performance degrades for larger k. An important drawback of this data structure is that it is not space efficient because it uses $O(k^2)$ space to store the arrangement. We now give a Monte Carlo data structure that supports insert and delete operations in sublinear time. The data structure relies on spanning trees of low stabbing number and the data structure from Section 3.1.

Theorem 12 *Geometric set partitions can be maintained with reports in $O(n)$ and insert/delete operations in $O(\sqrt{n}\log n)$ time after $O(n^{1.5}\log n)$ preprocessing time. The space usage is $O((n+k)\log n)$ bits or $O(n+k)$ machine words. Reports give correct results with high probability.*

Proof: First we preprocess the n points to obtain a spanning tree of low stabbing number. The stabbing number of a tree is the maximum number of intersections it can have with any line. We can find a spanning tree T that has a stabbing number of $O(\sqrt{n})$ in $O(n^{1.5}\log n)$ time [9, 28]. We *orient* each edge of T arbitrarily, and we maintain a color, initially 0, for the edge. We also assign a separate color for each partitioning line. On report, we use line and edge colors to assign colors to each node; see Section 3.1 for the definition of color.

To insert a line L we first assign a color to L and arbitrarily orient the line in one of the two directions. Then we find the set E of $O(\sqrt{n})$ edges of the tree T stabbed by L. For each edge e in E we either add or subtract the color of L modulo R from the color of e. We add if e goes from left to right of L and subtract otherwise. Finding the set E takes $O(\sqrt{n}\log n)$ time [9], and hence insertions take $O(\sqrt{n}\log n)$ time.

To delete a line L we find the set E of $O(\sqrt{n})$ edges of the tree T stabbed by L. Then we subtract or add modulo R the color of L from the color of each edge e of the set E. We subtract if e goes from left to right of L and add otherwise. As for insertions, finding the set E and hence deletion takes $O(\sqrt{n}\log n)$ time.

To report, we start with any node of T, which we designate as the root node, and we assign it the color 0. We then traverse the tree, e.g., with breadth-first search, and we assign each (non-root) node in the traversal a color based on the color of its parent node. For a node j, let $e = (i, j)$ be the edge between j and its parent i. Let c_e be the edge color of e. If e is oriented from i to j, assign the color $C[j] = C[i] + c_e \bmod R$ to node j otherwise if e is oriented from j to i, assign the color $C[j] = C[i] - c_e \bmod R$ to j. After assigning each node with a color, we report as in Section 3.1.

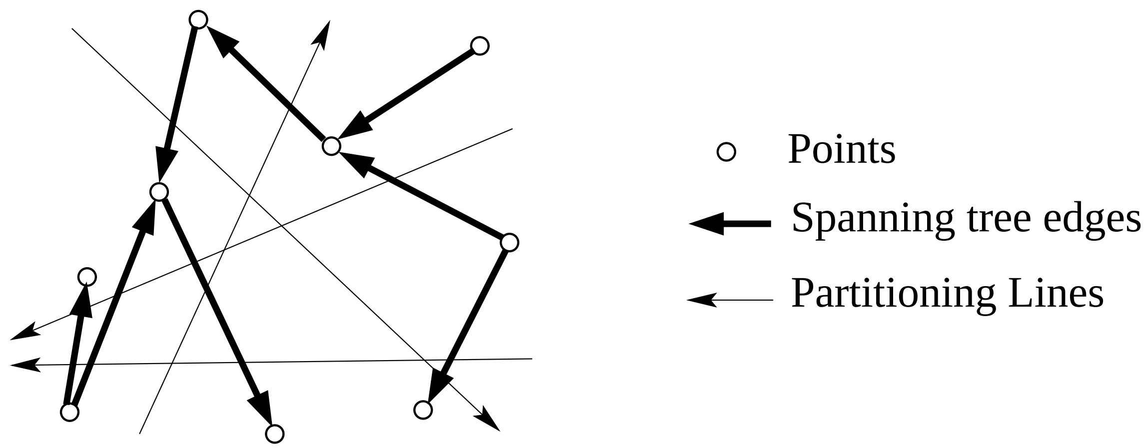

Figure 2: Monte Carlo algorithm for geometric set partitioning.

Correctness follows because for two points in the same cell of the arrangement of lines, the unique path in T between them intersects any line an even number of times. See Figure 2. Thus, while moving from one point to another we add and subtract colors of the lines an equal number of times and hence points in the same cell have the same color. On the other hand, if two points are in different cells, the unique path in T between them intersects at least one line an odd number of times. Hence, with high probability the two points have different colors. The error probabilities are as in Theorem 3.

The space usage follows because each line can be stored in $O(1)$ machine words, and we maintain $O(n + k)$ colors. ∎

Although the off-line version of constructing the induced set partition of k set partitions of n elements can easily be solved in optimal $\Theta(kn)$ time by repeated application of Lemma 1, the geometric version of the off-line problem is more interesting. Here we are given a set of k lines and n points, and we seek to determine their induced set partition. This problem can be solved using the well studied problem of computing many faces in arrangements. Agarwal [1] gives a deterministic algorithm with running time $O(n^{2/3}k^{2/3}\log^{O(1)} k + k\log^3 k + n\log k)$. Simpler randomized algorithms with similar running times are given by Agarwal, Matoušek and Schwarzkopf in [2].

Acknowledgments

We thank Alex Zelikovsky for introducing us to this problem. We thank the anonymous referees for comments that have greatly improved the presentation of this paper.

References

[1] P. K. Agarwal. Partitioning arrangements of lines: II. Applications. *Discrete Comput. Geom.*, 5:533–573, 1990.

[2] P. K. Agarwal, J. Matoušek, and O. Schwarzkopf. Computing many faces in arrangements of lines and segments. *SIAM J. Comput.*, 27(2):491–505, 1998.

[3] E. Arkin, H. Meijer, J. S. B. Mitchell, D. Rappaport, and S. Skiena. Decision trees for geometric objects. *Int. J. Computational Geometry and Applications*, 8:343–363, 1998.

[4] J.-P. Barthélemy and B. Leclerc. The median procedure for partitions. In I. J. Cox, P. Hansen, and B. Julesz, editors, *Partitioning Data Sets*, volume 19 of *DIMACS Series in Descrete Mathematics*, pages 3–34. American Mathematical Society, Providence, RI, USA, 1995.

[5] P. Belleville and T. C. Shermer. Probing polygons minimally is hard. *Comput. Geom. Theory Appl.*, 2(5):255–265, Mar. 1993.

[6] M. A. Bender, S. Sethia, and S. Skiena. Data structures for maintaining set partitions. In M. M. Halldórsson, editor, *Proc. The 7th Scandinavian Workshop on Algorithm Theory (SWAT)*, volume 1851 of *LNCS*, pages 83–96. Springer-Verlag, July 2000.

[7] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Wadsworth Inc., Belmont, CA, USA, 1984.

[8] G. Calinescu. A data structure for maintaining a partition. Manuscript, 2000.

[9] B. Chazelle, H. Edelsbrunner, M. Grigni, L. J. Guibas, J. Hershberger, M. Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12(1):54–68, July 1994.

[10] P. A. Chou. Optimal partitioning for classification and regression trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(4):340–354, 1991.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press and McGraw-Hill, 2nd edition, 2001.

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

[13] P. Dietz, K. Mehlhorn, R. Raman, and C. Uhrig. Lower bounds for set intersection queries. In *Proc. 4th ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 194–201, Austin, TX, USA, Jan. 1993.

[14] J. Feigenbaum and S. Kannan. Dynamic graph algorithms. In K. Rosen, editor, *Handbook of Discrete and Combinatorial Mathematics*, pages 583–591. CRC Press, Boca Raton, FL, USA, 1995.

[15] W. Feller. *An introduction to probability theory and its applications*, volume one. John Wiley & Sons, New York, NY, USA, 3rd edition, 1968.

[16] E. B. Fowlkes and C. L. Mallows. A method for comparing two hierarchical clusterings. *Journal of the American Statistical Association*, 78(383):553–584, Sept. 1983.

[17] R. Freimer, J. S. B. Mitchell, and C. D. Piatko. On the complexity of shattering using arrangements. In *Proc. 2nd Canad. Conf. Comput. Geom. (CCCG)*, pages 218–222, Aug. 1990.

[18] R. Freimer, J. S. B. Mitchell, and C. D. Piatko. On the complexity of shattering using arrangements. Technical Report TR 91-1197, Dept. Comput. Sci., Cornell Univ., Ithaca, NY, USA, Apr. 1991.

[19] M. R. Garey. Optimal binary identification procedures. *SIAM J. Appl. Math.*, 23(2):173–186, 1972.

[20] L. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. Foundations of Computer Science*, pages 8–21, Ann Arbor, MI, USA, 1978. Springer-Verlag.

[21] M. Habib, C. Paul, and L. Viennot. A synthesis on partition refinement: A useful routine for strings, graphs, boolean matrices and automata. In M. Morvan, C. Meinel, and D. Krob, editors, *Proc. 15th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1373 of *LNCS*, pages 25–38. Springer-Verlag, Feb. 1998.

[22] J. Hopcroft. An $n \log n$ algorithm for minimizing the states in a finite automation. In Z. Kohavi, editor, *The theory of machines and computations*, pages 189–196. Academic Press, New York, NY, USA, 1971.

[23] L. Hubert and P. Arabie. Comparing partitions. *Journal of Classification*, 2:193–218, 1985.

[24] L. Hyafil and R. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, May 1976.

[25] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall, NJ, USA, 1988.

[26] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley, Reading, MA, USA, 1st edition, 1973. See page 466.

[27] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* B. G. Teubner, Stuttgart, Germany, July 1990.

[28] J. Matoušek. More on cutting arrangements and spanning trees with low crossing number. Technical Report B-90-2, Fachbereich Mathematik, Freie Universität Berlin, Berlin, Germany, 1990.

[29] B. Moret. Decision trees and diagrams. *Computing Surveys*, 14(4):593–623, Dec. 1982.

[30] R. Motwani and P. Raghavan. *Randomized Algorithms.* Cambridge University Press, Cambridge, England, June 1995.

[31] J.-M. Muller. *Arithmétique des ordinateurs.* John Wiley & Sons, Paris, France, 1989.

[32] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Computing*, 16(6):973–989, Dec. 1987.

[33] W. M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, Dec. 1971.

[34] S. R. Safavian and D. Landgrebe. A survey of decision tree classifier methodology. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(3):660–674, 1991.

[35] G. Sazaklis, E. Arkin, J. S. B. Mitchell, and S. Skiena. Geometric decision trees for optical character recognition. In *Proc. of 13th Annual ACM Symposium on Computational Geometry (SOCG)*, pages 490–492, June 1997.

[36] G. Sazaklis, E. Arkin, J. S. B. Mitchell, and S. Skiena. Probe trees for touching character recognition. In *Proc. International Conference on Imaging Science, Systems and Technology, (CISST)*, pages 282–289, Las Vegas, NV, USA, July 1998.

[37] S. Skiena. Interactive reconstruction via probing. *IEEE Proceedings*, 80:1364–1383, 1992.

[38] R. Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, Apr. 1975.

[39] Y. Wakabayashi. The complexity of computing medians of relations. *RESENHAS, Instituto de Matemática e Estatística, Universidade de São Paulo (IME-USP), São Paulo - SP, Brasil*, 3(3):323–349, 1998.

[40] D. Yellin. Representing sets with constant time equality testing. In *Proc. 1st ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 64–73, Jan. 1990.

[41] D. Yellin. Algorithms for subset testing and finding maximal sets. In *Proc. 3rd ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 386–392, Jan. 1992.