Efficient Data Structures for Maintaining Set Partitions (Extended Abstract)

Michael Bender Saurabh Sethia Steven Skiena

Department of Computer Science
State University of New York
Stony Brook, NY 11794-4400
{bender—saurabh—skiena}@cs.sunysb.edu

April 22, 1999

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 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 [20, 21] based on point-probe decision trees [1], 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.

In this paper, we introduce techniques to speed up this process of feature identification. We propose a series of a 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 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.

Previous Work. A considerable literature on the subject of classification and regression trees exists; see [3, 5, 17, 19]. The abstract decision tree problem takes as input a universal $U = \{1, \ldots, n\}$ 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 [15] prove that it is NP-complete to construct a minimum height or a minimum external path length decision tree. Garey [12] presents dynamic programming algorithms for determining an optimal weighted decision tree. It is is NP-complete to identify the minimum set of features sufficient to build a decision tree even in very restricted geometric instances [2, 11]. See [22] 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 none seem 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 [23] 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). Yellin [26] efficiently supports a variety of subset testing operations (insert/delete elements, create subsets, subset and intersection queries) all in $O(n^{1/2} \log n)$ time per operation, but it is not clear how to use these operations to improve even the naive bounds for our problem. Further, near matching lower bounds are known on the complexity of any data structure which supports these operations [8].

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 [10], 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.

Our Results. In this paper, we present a collection of efficient and practical data structures for maintaining set partitions, as well as several generalizations of the problem. Particularly interesting is the variety of algorithmic techniques which they encompass, including classical balanced trees, randomization, suffix trees supporting least common ancestor queries, and spanning trees of low stabbing number. In particular:

- We provide a data structure which supports the insert and report operation in optimal $O(n)$ worst-case time, for general set partitions. Deletion takes $O(n \lg k)$ time, where k is the number of set partitions currently in the data structure and n is the number of elements in each partition.

- We provide randomized Monte Carlo and Las Vegas data structures which support all three operations on bipartitions in linear or near-linear expected time, although the Las Vegas bounds are amortized. We believe that our Monte Carlo algorithm is particularly practical.

- Robust classifiers compensate for noisy features by requiring more than one piece of evidence to distinguish between every pair of objects. We provide an alternate data structure that permits us to efficiently insert/delete partitions and query arbitrary pairs of elements $\{x, y\}$ to report the approximate number of partitions currently distinguishing x from y. This data structure uses techniques from random walks on a line.

- We provide an efficient 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.

- We provide the first data structures for efficiently maintaining sorted strings under character insertion/deletion. As an application, we use this structure to find the shortest run of distiguishing features from an ordering of k binary features in optimal $O(nk)$ time. Our data structure, based on suffix trees, is also of independent interest.

- We investigate the related problem of sorting $k \times n$ binary matrices through row and column permutations so both the rows and columns are simultaneously sorted. We prove that such an ordering always exists, and give an $O(nk \min(k, n))$ algorithm to produce such an ordering.

Our paper is organized as follows. In Section 2, we present deterministic data structures for maintaining set partitions. More efficient randomized data structures are presented in Section 3. The problem of maintaining robust classifers is discussed in Section 4. The special case of set partitions induced by geometric arrangements is discussed in Section 5. A generalization of our problem, sorting strings under character insertion/deletion is addressed in Section 6. Finally, we consider an interesting related combinatorial problem on binary matricies in Section 7.

2 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 retreiving these pointers to an auxilliary dictionary data structure.

Notation. We use the following notation throughout the paper. Let the universal set $U = \{1, \ldots, n\}$. Each set partition P partitions U into parts(P) disjoint subsets $P_1, \ldots, P_{\text{parts}(P)}$ such that $\cup_{i=1}^{parts(P)} P_i = U$. Without loss of generality, we identify these subsets by the integers $(1, \ldots, \text{parts}(P))$, respectively. Let part$(P, i)$ denote the part of P containing element i.

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 $(\text{part}(A, i), \text{part}(B, i), i)$. All pairs of strings whose first two elements are identical represent elements who 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 insertion and report in linear time but does not explicitly support deletion. A naive 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 $O(n \lg k)$ time, while report can be performed in $O(n)$ time.*

Proof. We maintain a balanced binary tree 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 this tree 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 done in any balanced binary tree such as [13]. Insertion and deletion 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 $O(\lg k)$ intermediate root-to-leaf nodes must be recomputed using Lemma 1, so that the time required for insertion and/or deletion is $O(n \lg k)$. ∎

We note that a similar structure, called a partition tree, appears in a different context in Yellin [25]. We can modify the data structure of Lemma 2 to reduce the complexity of insertion to linear:

Theorem 1 *Set partition can be maintained with $O(n)$ insertion and report, and $O(n \lg k)$ deletion.*

Proof. Instead of maintaining a conventionally-balanced binary tree in the structure of Lemma 2, we maintain a forest of perfectly-balanced binary trees.

As before, the leaf level of our forest contains all k of the input partitions. We number them from 1 to k according to time of insertion. On each insertion, we will add one leaf to one tree, and construct at most one additional internal node. Denoting where the leaves reside as level 0, we will add a new internal node at the ith level every 2^ith insertion. For each level i, we will maintain a FIFO queue of pointers to the roots of trees of height i. In addition to this forest, we will also maintain a separate global induced partition, initially $\{1, \ldots, n\}$.

Inserting a Partition: We insert the kth partition as follows:

1. Increment the partition counter k. Construct a new leaf node for partition P_k. Add the node k to the end of the level 0 queue.
2. Refine the global set partition with P_k using the algorithm of Lemma 1.
3. Define j to be the largest integer such that $k+1 \geq 2^j$. Compute b, the position of the least significant 1-bit of the binary representation of $s = k - 1 - 2^j$. If $s = 0$, then b is undefined.
4. Unless $s = 0$, dequeue the two oldest elements A and B of level queue b. Merge the associated set partitions of A and B using the algorithm of Lemma 1. Construct a new internal node to contain the refined partition of A and B, and enqueue this node at level $b+1$.

To implement report, we simply return the global result partition. To implement deletion, we replace the partition to be deleted by the partition of the last leaf to have been inserted, and delete the internal node (if any) constructed during the insertion of this leaf. We then recompute the $O(\lg k)$ induced partitions of the internal nodes on the two effected root-to-leaf paths. Finally, we compute the new global result partition by merging all the $O(\lg k)$ root partitions in our forest. Hence it takes $O(n \lg k)$ time to do a deletion.

Note that the structure of forest depends only on the number of partitions currently in S, and is independent of any deletions which may have taken place. Hence to measure the complexity of insertions we can safely assume that k insertions were performed sequentially. ∎

3 Randomized Data Structures for Maintaining Set Partitions

Assumptions. For simplicity we assume that each partition that we insert is a *bipartition*, meaning it divides the elements into exactly two subsets. The results in this section can be generalized to the setting in which a partition breaks the elements into D sets, but at an extra cost proportional to D.

We say that an event E occurs *with high probability* (*w.h.p.*) if for any $c > 0$ there exists a proper choice of constants such that $\Pr[E] \geq 1 - n^{-c}$.

3.1 Monte Carlo Algorithm

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 $C_t[i]$ the *color* of element i at step t. Specifically, $C_t[i] \in \{0, \ldots, P-1\}$, where P has size polynomial in n. That is, $P \in O(n^c)$, for some constant c. We maintain the invariant that w.h.p., if two elements i and j are in the same set at step t in the (cumulative) partition S iff they have the same color, that is, $C_t[i] = C_t[j]$.

We also store the partitions $P_1 \ldots P_K$ that comprise S, where the partitions are ordered by increasing insertion time. We can access any element in the set of partitions in time $O(\log K) \in O(n)$ using a balanced tree or any other basic data structure.

Inserting a Partition. A new partition is supplied as a 0-1 array $A[1 \ldots n]$, where $A[i] \in \{0, 1\}$. We *insert* the k-th partition in step t as follows.

1. $r_k :=$ randomly chosen integer $\in \{1, \ldots, P-1\}$.
2. $P_k[i] := r_k \cdot A_k[i]$, for $i = 1 \ldots n$.
3. $C_t[i] := C_{t-1}[i] + P_k[i]$ mod P, for $i = 1 \ldots n$.
4. Store $P_k[1 \ldots n]$ in the list of partitions.

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

Deleting a Partition. We *delete* a partition P_k (in step t) as follows.

1. Find $P_k[1 \ldots n]$ in the partition list.
2. $C_t[i] := C_{t-1}[i] - P_k[i] \bmod \mathrm{P}$, for $i = 1 \ldots n$.
3. Delete $P_k[1 \ldots n]$ from the partition list.

Time Complexity. We now analyze the time complexity of insertions and deletions.

Theorem 2 *Insertions and deletions of partitions are executed in time $O(n)$.*

Proof. In an insertion, step 1 requires $O(\log P) \in O(\log n)$ random bits. In practice this entails one call to a random number generator. Steps 2 and 3 each requires time $O(n)$. Step 4 requires time $O(\log K) \in O(n)$. (This follows because there are at most 2^n different subsets of n numbers.)

In a deletion, steps 1 and 3 each require $O(\log K) \in O(n)$ time. Step 2 requires $O(n)$ time. ∎

Probability of Error. We now provide a bound on the probability of error. Observe that errors are one-sided. Namely, if two elements have different colors, they belong to different sets of S. An error occurs whenever two elements assigned the same color actually belong to different sets.

Theorem 3 *If the algorithm runs for a polynomial number of steps, then for sufficiently large $P = O(n^c)$, w.h.p. all insertions and deletions are executed correctly.*

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 are assigned the same color but belong to different sets. More formally, t is a bad step for elements i and j iff

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

but erroneously,

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

In each step t, a partition $P_k[1 \ldots n]$ is either inserted or deleted. Assume w.l.o.g. that $P_k[i] = 0$ and that $P_k[j] = r_k$. On an insertion, there is a bad step for i and j iff

$$r_k = C_{t-1}[i] - C_{t-1}[j] \mod P.$$

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

$$r_k = C_{t-1}[j] - C_{t-1}[i] \mod P.$$

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

We note that the Monte Carlo algorithm should be extremely fast because it only uses a small number of additions and subtractions. It is interesting to note that our Monte Carlo algorithm 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. Since absolutely minimizing the number of probes in a test set is NP-complete even in restricted geometric instances [2, 11], this penalty is particularly excusable.

3.2 Las Vegas Algorithm

We now describe the Las Vegas algorithm for the set partition problem. The time complexity is almost the same as for the Monte Carlo algorithm except that now it is *amortized*. In order to make the algorithm Las Vegas, we remove the probability of error from the invariant that whenever two elements have the same color, they are part of the same set in S. Thus, each time we perform an insertion or deletion, we must *verify* that this invariant holds.

Verifying an Insertion. We first show how to verify an insertion for each step t. This entails adding an additional step at the end of the operation:

5. Verify that for all i, j, $(\, C_t[i] = C_t[j]\,) \Longrightarrow (\, C_{t-1}[i] = C_{t-1}[j]\,)$
6. If so, continue. If not, an *error* is found. Spawn off an independent execution or run any other alternative protocol.

Step 5 can be executed in linear time by maintaining some additional structure of the elements. Namely, in each step we will keep the elements in sorted order by increasing color. To do so, we use an additional array $\Pi_t[1\ldots n] = \pi_1^{(t)} \ldots \pi_n^{(t)}$, where $\pi_\ell^{(t)} = j$ means that in step t, j is the element with the ℓ-th smallest color. (Ties are broken by taking into account the number of the element.) 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,
- the elements of $\Pi_t[1\ldots n]$ whose colors increase by r_k and then (by the rules of modular arithmetic) decrease by P.

Thus, step 5 can be broken into these substeps:

5a. Compute $\pi_1^{(t)} \ldots \pi_n^{(t)}$ from $\pi_1^{(t-1)} \ldots \pi_n^{(t-1)}$ by merging three lists.

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

$$\left(C_t[\pi_\ell^{(t)}] = C_t[\pi_{\ell+1}^{(t)}]\right) \Longrightarrow \left(C_{t-1}[\pi_\ell^{(t)}] = C_{t-1}[\pi_{\ell+1}^{(t)}]\right)$$

This test requires linear time because we only have to compare the color of each element $\pi_\ell^{(t)}$ with its neighboring elements $\pi_{\ell-1}^{(t)}$ and $\pi_{\ell+1}^{(t)}$ in the ordering.

Verifying a Deletion. We now show how to verify the deletion of partition $P_u[1\ldots n]$ in step t. Let $P_{x_1} \ldots P_{x_m}$ be the existing partitions in the system after $P_u[1\ldots n]$ is deleted. Let P_{x_k} be the partition that appeared in S just before P_u.

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 separating i and j has been removed, or whether i and j are erroneously assigned the same color and are in fact separated by many partitions. Thus we verify deletions as follows.

4. Verify that if $C_t[i] = C_t[j]$, then i and j are in the same set of all partitions $P_{x_1} \ldots P_{x_n}$ (except for $P_u[1 \ldots n]$).
5. If so, continue. If not, an *error* is found. Spawn off an independent execution or run any other alternative protocol.

We now show how to make the verification efficient. Note that step 4 could potentially be very expensive because it may involve scanning through the entire list of partitions $P_{x_1} \ldots P_{x_n}$. Despite this, we show that the amortized cost of verifying a deletion is $O(n\alpha(n))$, where $\alpha(n)$ is the inverse Ackerman function. To do this we will 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 maintain a UNION-FIND data structure for each prefix of partitions $P_{x_1} \ldots P_{x_m}$. Thus, two elements i and j belong to the same set in the data structure $\text{UNION-FIND}^{(x_\ell)}$ if they belong to the same set in all of the partitions $P_{x_1} \ldots P_{x_\ell}$. Because partitions are always inserted *at the end* of S, the sets in each $\text{UNION-FIND}^{(x_\ell)}$ data structure may coalesce when partitions are deleted but will never split apart. The sets can combine together at most $n-1$ times before a single set remains and consequently all elements have the same color. The operation $\text{FIND-SET}^{(x_\ell)}[i]$ locates the *smallest* element belonging to the same set as element i (in the $\text{UNION-FIND}^{(x_\ell)}$ data structure). The operation $\text{UNION}^{(x_\ell)}[i\,,\,j]$ *combines* the set containing i and the set containing j (in the $\text{UNION-FIND}^{(x_\ell)}$ data structure).

Let $C^{(x_\ell)}[i]$ represent the color of element i after inserting partitions $P_{x_1} \ldots P_{x_\ell}$, for $\ell = 1 \ldots n$.

4a. Compute $\pi_1^{(t)} \ldots \pi_n^{(t)}$ from $\pi_1^{(t-1)} \ldots \pi_n^{(t-1)}$ by merging three lists.

4b. By incrementing the value of ℓ, find the first ℓ such that $C_t[\pi_\ell^{(t)}] = C_t[\pi_{\ell+1}^{(t)}]$, but

$$\text{FIND-SET}^{(x_m)}[\pi_\ell^{(t)}] \neq \text{FIND-SET}^{(x_m)}[\pi_{\ell+1}^{(t)}]\,.$$

(This means that $C_{t-1}[\pi_\ell^{(t)}] \neq C_{t-1}[\pi_{\ell+1}^{(t)}]$.)

4c. For such a value of ℓ, we execute the following:

- Verify that
$$C^{(x_k)}[\pi_\ell^{(t)}] = C^{(x_k)}[\pi_{\ell+1}^{(t)}]$$
(Recall that P_{x_k} is the partition in S just before the partition P_u that is deleted.) This can be verified in constant time by checking that
$$\text{FIND-SET}^{(x_k)}[\pi_\ell^{(t)}] = \text{FIND-SET}^{(x_k)}[\pi_{\ell+1}^{(t)}]\,.$$
If this check fails, then an *error* is found.
- Verify that for all partitions $P_x = P_{x_{k+1}} \ldots P_{x_m}$,
$$C^{(x)}[\pi_\ell^{(t)}] = C^{(x)}[\pi_{\ell+1}^{(t)}]\,.$$
If this check fails, then an *error* is found.
- Otherwise, for all partitions $P_x = P_{x_{k+1}} \ldots P_{x_m}$,
$$\text{UNION}^{(x)}[\pi_\ell^{(t)}\,,\,\pi_{\ell+1}^{(t)}]$$

Theorem 4 *Verifying an insertion requires time $O(n)$. Verifying a deletion requires amortized time $O(n\alpha(n))$.*

Proof. For verifying insertions, step 5a requires linear time because it entails merging three lists.

For verifying deletions, step 4a requires linear time for the same reason. Step 4b requires linear time because each check takes constant time and there are $n-1$ checks. Step 4c requires amortized $O(\alpha(n))$ per partition per set partition since there are at most $n-1$ set unions per partition, the amortized cost of $O(n\alpha(n))$ follows. ∎

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 theorem.

Theorem 5 *Insertions run in $O(n)$ time, both expected and w.h.p.. Deletions run in amortized time $O(n\alpha(n))$, both expected and w.h.p..*

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. In particular, for fault tolerance, one may insist that each element be separated by at least k 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 naive solution to this problem, where the data structure maintains a two-dimensional array Count$[1 \ldots x][1 \ldots x]$, where Count$[i][j] = k$ signifies that k 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.

Our data structure supports the following three operations.

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

We show that the operations *insert* and *delete* can be implemented to run in worst-case time $O(n \log n)$, or expected $O(n)$ time. *Query* requires polylogarithmic time and returns an answer that is accurate to within any constant factor. As it is stated, this algorithm works only for bipartitions.

As in Section 3, we assign integers called *colors* to elements. However now the *distance* between two colors *approximates* the number of partitions separating elements. More specifically, each element i has $\beta \log n$ separate colors, C$[i][1 \ldots \beta \log n]$. Each color is modified independently.

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

- Randomly choose one of the two representations of the partitions for each of the $\beta \log n$, and
- add these values to the composite colors of the elements.

Thus in the i-th position some of the colors are incremented by 1 and some of the colors remain the same.

Deleting a partition. To delete a partition, we again modify each of the $\beta \log n$ colors inversely to the changes on inserting the partition. For each color, we subtract the appropriate representations of the partition from the composite color so that each color either remains unmodified or decreases by 1.

Querying elements i and j. To *estimate* the number of colors separating elements i and j, we compare the colors of i and j. Let K_t be the number of partitions in the system at time t, and let $E(\,\mathrm{C}[i][l]\,)$ be the expected value of $\mathrm{C}[i][l]$.

Notice that,

$$\begin{aligned} E(\mathrm{C}[i][1]) \;=\; E(\mathrm{C}[i][2]) \;=\; \ldots \;=\; E(\mathrm{C}[i][\beta \log n]) \;=\; \\ E(\mathrm{C}[j][1]) \;=\; E(\mathrm{C}[j][2]) \;=\; \ldots = E(\mathrm{C}[j][\beta \log n]) \;=\; K_t/2. \end{aligned}$$

However the actual colors will *deviate* from this expected value. If no sets separate i and j, then for all $\ell = 1 \ldots \beta \log n$, $\mathrm{C}[i][\ell] = \mathrm{C}[j][\ell]$. The less the value of $\mathrm{C}[i][\ell]$ and $\mathrm{C}[j][\ell]$ are correlated, the more sets separate i and j. Specifically, if there are d sets that separate elements i and j, then we can view the process of choosing colors for i and j as a *random walk* of length d in the following sense.

Consider the most basic random walk on the integer line. A walk starts at the origin and at each step t, moves one unit to the right with probability $1/2$ and moves one unit to the left with probability $1/2$. We compare this random walk with the dynamics of the algorithm. Whenever i and j are in different partitions than with probability $1/2$, the color of i is incremented by 1 and the color of j stays the same, and with probability $1/2$, the color of j is incremented by 1 and the color of i stays the same.

Thus, the probability

$$\Pr\left[\,\mathrm{C}[i][\ell] - \mathrm{C}[j][\ell] = z\,\right]$$

is exactly the probability that a random walk of length d ends at integer z. Thus, by examining the distribution of the colors of the elements, we can estimate the most likely value of d. In the estimation method we use the language of random walks. We can estimate d to within any constant factor because of the following theorem.

Theorem 6 *Consider the probability that a random walk of length d ends at a distance Δ from the origin, where*

$$\alpha_1 \sqrt{d} \leq \Delta \leq \alpha_2 \sqrt{d}.$$

As $d \to \infty$, this probability approaches the value $\frac{2}{\sqrt{\pi}} \int_{\alpha_1}^{\alpha_2} e^{-2\,t^2}\, dt$, which is independent of d. For small d, lower order terms can be included to achieve additional accuracy.

Estimation Method. Let $\textsc{expected-inside}(x)$ be the expected number of random walks that end within x of the origin in a walk of length x^2. Let $\textsc{inside}(x)$ be the number of walks that end within x of the origin in our algorithm. We estimate d by finding the largest integer d' such that

$$\textsc{inside}(\sqrt{d'}) \geq \textsc{expected-inside}(\sqrt{d'}).$$

Thus, we say that d is approximately d' or $d'+1$. (We expect that maximum likelihood estimation of d also yields a good estimate of d, but the above algorithm lends itself to straightforward error analysis.)

Theorem 7 *For any error parameter ϵ and constant c, there is a constant β such that the following holds with probability $1 - 1/n^c$ (w.h.p.): If the number of sets separating elements i and j is d, then the estimate d' of d is bounded as follows:*

$$(1-\epsilon)d \leq d' \leq (1+\epsilon)d.$$

Proof. We provide a bound on the probability that $d' < (1-\epsilon)d$ or $d' > (1+\epsilon)d$. Here we bound the probability that the first inequality holds. Let X_i be the $0/1$ random variable where

$$X_i = \begin{cases} 1 & \text{if a random walk of length } d \text{ ends within } \sqrt{(1-\epsilon)\,d}, \\ 0 & \text{otherwise.} \end{cases}$$

Thus $X = X_1 + X_2 + \ldots + X_{\beta \log n}$ is the number of walks that end within $\sqrt{(1-\epsilon)\,d}$. Note that by Theorem 6, the expected number of random walks that end within $\sqrt{(1-\epsilon)\,d}$ is a constant factor smaller than the expected number of random walks that end within $\sqrt{d}$ of the origin. Thus, using Chernoff bounds (e.g., from [18]), we can show that $\Pr[X \geq \text{EXPECTED-INSIDE}(x)]$ is polynomially small for sufficiently large β. A similar argument shows that $d' > (1+\epsilon)d$. ∎

To implement the insert and delete operations in expected $O(n \log\log n)$ time, we must show how to increment the $\beta \log n$ colors by the prescribed random bits in expected $\log\log n$ time. This can be done by maintaining these colors as $\log k$ sets of 2β integers, each of size $\log n$. The first set of integers contains the least significant bit of $(\log n)/2$ colors, with each data bit flanked by zero bits. Adding the random bits (similarly padded) to this integer increments each of the colors simulataneously. If any carries occur, they appear as 1 bits in the padded region, and require us to increment the appropriate next significant bits. Each round of incrementing takes constant time, and the expected number of levels to propegate the carries is $\log\log n$.

5 Maintaining Geometric Set Partitions

Suppose the elements in each set partition were points in the plane, and that each set partition was induced by a half-plane that distinguishes between the points which lie to the left or right of the defining line. Such partitions have been previously studied. For example, Freimer, et.al [11] prove it is NP-complete to find the smallest subset of lines sufficient to completely 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 half-planes against each point and reducing it to a non-geometric instance. However, exploiting the geometry can make the problem easier. The following operations should be supported by such a data structure.

- *Insert-line(l,S)* – add a separating line l to the arrangment of partitions/points S.
- *Delete-line(l,S)* – delete existing separating line l from the arrangment of partitions/points S.
- *Report(S)* – report the set partition of U induced by the arrangment of partitions/points S.
- *Query(p, q, S)* – determine whether point p and q are in the same component of the set partition of U induced by the arrangment of partitions/points S.

A naive way to support these queries is by maintaining the arrangement of the halfplanes. Any two points in the same cell of the arrangment represent unpartitioned elements. A line insertion/deletion into the arrangment takes $O(k+n)$ time, the former term for inserting a line in an arrangement of k lines [7] and the latter term to partition the lists of points in the split. And 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. And query takes constant time because each point knows the cell to which it belongs.

This naive algorithm is faster than the algorithms in the non-geometric setting when k is $O(n)$, but its performance degrades for larger k. An important drawback of this algorithm is that its not space efficient because it uses $O(k^2)$ space to store the arrangement. We improve this naive algorithm so that insert and delete run in sublinear time. The algorithm uses spanning trees of low stabbing number and randomization.

Theorem 8 *Geometric set partitions can be maintained with report in $O(n)$, insert/delete in $O(\sqrt{n}\log n)$, and query in $(\sqrt{n})$ time. The data structure uses $O(n+k)$ space and $O(n^{1.5}\log n)$ preprocessing time. The algorithm runs correctly in polynomial time w.h.p.*

Proof. First we preprocess the n points to obtain a spanning tree of low stabbing number. We can find a spanning tree T of stabbing number of $O(\sqrt{n})$ in $O(n^{1.5}\log n)$ time [4, 16]. We *orient* each edge of T arbitrarily. In addition we maintain an integer or *color* for each edge, initially 0.

To insert a line L we first associate an integer or *color* with L. As in the non-geometric randomized algorithms, the color is a randomly chosen integer between 1 and $P-1$, where P is $O(n^c)$ for some constant c. Then we find the set E of $O(\sqrt{n})$ edges of tree T stabbed by L. For each edge e in E we either add of subtract the color of L modulo P 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 [4], 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 tree T stabbed by L. The we subtract or add modulo P 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 E and deletion takes $O(\sqrt{n}\log n)$ time.

To report we start with any node of T and assign it an integer or *color* 0. We then traverse the tree and assign each node of T a color as follows. If a node has color c_0 and has an outgoing edge e_1 with color c_1 then the target of e gets a color $c_0 + c_1 \bmod (P)$. Similarly for an incoming edge e_2 with color c_2 the source node gets a color $c_0 - c_1 \bmod (P)$. After assigning each node with a color, we radix sort them by color and report the partition of nodes induced by their colors. Clearly, this can be performed in $O(n)$ time.

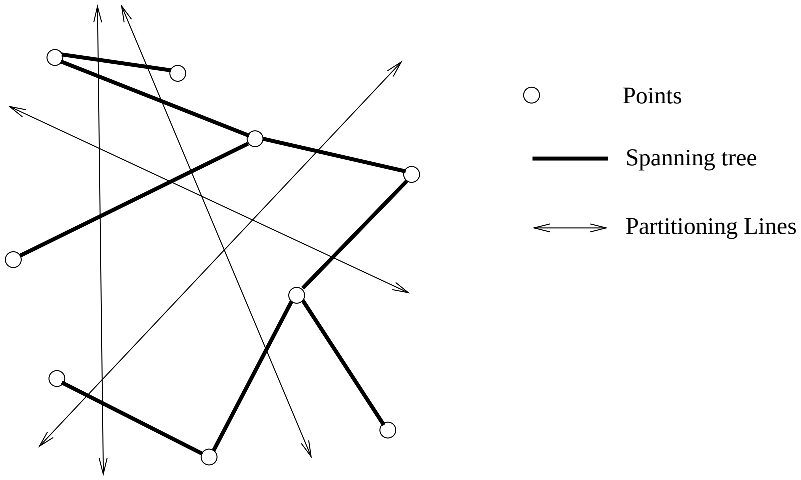

Figure 1: Monte-Carlo algorithm for geometric set partitioning.

Note that this works because for two points in the same cell of arrangement of lines, the unique path in T between them will intersect any line an even number of times. See Figure 1. Thus if while moving from one point to another we would add and subtract colors of the lines an equal number of times and hence points in the same cell will have same color. On the other hand, if two points are in different cells the unique path in T between them would intersect at least one line odd number of times. Hence w.h.p. the two points will have different colors.

Details appear in the full paper. ∎

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 seek to determine the induced set partition of them.

Theorem 9 *The induced set partition of k lines and n points in the plane can be determined in total $O(n^{1.5} + k\sqrt{n})$ time and $O(n + k)$ space w.h.p.*

Theorem 10 *The induced set partition of k lines and n points in the plane can be determined in total $O(k^{1.5}\log^{\omega} k + n\sqrt{k}\log^2 k)$ time and $O(k\log^2 k)$ space, where ω is a constant less than 4.3.*

6 Maintaining Sorted Strings under Character Insertion/Deletion

The problem of determining the partition induced by a collection of k set partitions on $\{1, \ldots, n\}$ can easily be reduced to that of sorting strings. Arbitrarily assign each of the parts of each partition p a distinct number from 0 to $parts(p) - 1$. Define S_i to be the string of parts associated with element i, i.e. $S_i[j] = q$ iff element i is in part q of set partition j. Elements are x and y are indistinguishable iff $S_x = S_y$. Hence sorting strings $\{S_1, \ldots, S_n\}$ groups the elements into blocks of equivalence classes. This gives us an alternate, more general formulation which yields our original problem as a special case – maintain the sorted order of strings where we are (1) allowed to delete the ith character from each string and (2) append an extra character to each string.

In general, we have the problem of maintaining a set of strings $S = \{S_1, \ldots, S_n\}$ under the following operations:

- *Report(S)* – Return the permutation of S representing the the sorted order of the strings, with runs of duplicates identified.
- *Insert(S,i,T)* – Insert character $T[j]$ after the ith position each string S_j, $1 \leq j \leq n$. This increases the length of each string in S by one character.
- *Delete(S,i)* – Delete the ith character of each string S_j, $1 \leq j \leq n$. This decreases the length of each string in S by one character.

A data structure to implement these operations efficiently would yield a data structure for maintaining dynamic set partitions as a special case. The partition induced by each test can be inserted as a string at a given character position. If such strings are kept sorted with duplicates identified, the duplicates define the classes in the induced partition. The deletion of the ith test is equivalent to deleting ith character of each string. Since the order of tests is not important inserting a test can be implemented by appending a character to all strings. Thus our special case involves only tail insertions.

We propose a series of data structures based on suffix trees [14] to efficiently support a restricted set of these operations. In particular, we build our data structure around Ukkonen's linear time suffix tree construction algorithm [24]. In Ukkonen's algorithm, suffixes are inserted into the tree from left to right. Analogously, we can continue to append new characters onto the end of a string by simulating the insertion of another subsequent suffix.

We will augment this suffix tree to support constant-time least common ancestor queries. Cole and Hariharan [6] demonstrate how to maintain constant-time least common ancestor queries in trees supporting leaf insertion/deletion and edge-spliting updates.

Theorem 11 *A data structure can support the operations of head deletion, tail insertion, and sorted-order reporting in times $O(n)$, (amortized) $O(n)$, and $O(n+k)$ respectively.*

Proof. We construct a single implicit suffix tree containing all k suffixes of each of the strings $S_1, \ldots, S_n$. Let $S_i[j]$ denote the jth suffix of S_i, $1 \leq j \leq k$. Therefore this tree contains at most kn leaves, and we maintain a matrix of pointers allowing us constant time access to the suffix tree node associated with each $S_i[j]$.

On each suffix insertion operation, we simulate a phase of Ukkonen's algorithm by inserting the new, single character suffix $S_i[k+1]$ associated with each of the n strings. We update the least common ancestor structure to reflect these new nodes. The entire operation takes amortized $O(n)$ time – amortized because Ukkonen's algorithm does not ensure constant time per suffix insertion.

On a prefix deletion operation, we simply delete the n nodes cooresponding to the suffixes $S_i[1]$ for $1 \leq i \leq n$, and update the LCA structure accordingly.

To perform the sorted-order reporting in linear time $O(n+k)$, we use the Dietz-Sleator algorithm [9] which maintains the order of elements of a linked list so as to permit insertion, deletion, and order comparison (is x before y in the list?) each in constant time per operation. This data structure works by maintaining a small integer ($O(m)$, where m is the number of elements currently in the list) for each element, monotonically increasing with rank. An order comparison simply compares the two associated integers in constant time.

We will maintain (using the Dietz-Sleator algorithm) a linked list of all of the vertices in the suffix tree cooresponding the inorder traversal of tree. Since there are $O(kn)$ vertices in this tree, the integer associated with each vertex is $O(kn)$. We can sort the n nodes cooresponding to the full strings in the tree by sorting the in-order traversal numbers of these strings in $O(n+k)$ time by using a distribution/radix sort. ∎

The data structure of Theorem 11 can be used to efficiently obtain the shortest continguous discriminating run of k preordered tests, which suggests a new heuristic approach to finding small sets of discriminating

Initial Matrix					Column Permuted					Row Permuted				
rows	C_1	C_2	C_3	C_4	rows	C_3	C_4	C_1	C_2	rows	C_1	C_2	C_3	C_4
R_1	1	1	0	0	R_1	0	0	1	1	R_3	0	0	1	0
R_2	0	1	0	1	R_2	0	1	0	1	R_2	0	1	0	1
R_3	0	0	1	0	R_3	1	0	0	0	R_4	1	0	1	1
R_4	1	0	1	1	R_4	1	1	1	0	R_1	1	1	0	0

Figure 2: Two ways of double sorting a 0-1 matrix (on left).

features. We repeatedly insert new features in the given order until they first suffice to completely discriminate the n strings. At this point, we repeatedly delete the prefix characters of each string, until the refined partition contains fewer than n parts. By interleaving these phases of insertion and deletion, and maintaining the boundaries of the shortest discriminating run encountered along the way:

Corollary 12 *The shortest continguous discriminating run of k ordered tests can be computed in $O(k(n+k))$ time.*

The presence of arbitrary deletions significantly complicates the problem of maintaining sorted order. However, we have reasonable results when the number of deletions is small:

Theorem 13 *A data structure can support the operations of arbitrary deletion, tail insertion, and sorted-order reporting in times $O(n)$, (amortized) $O(n)$, and $O(dn \lg n)$ respectively, where d is the total number of deletions which have been performed.*

7 Double Sorting Binary Matricies

Let M be a 0-1 matrix with n rows and k columns. Each row or column of such a matrix can be interpreted as a binary number, where the most significant bit (MSB) in each row (column) lies in the leftmost (topmost) position. We seek to permute the rows and columns of M to generate a matrix M' which is simultaneously sorted in ascending order by both rows (top to bottom) and columns (left to right).

The relevance of this problem to set partition maintainence is as follows. Let each row represent an element of U, and each column a set partition. Sorting the rows groups all runs of indistinguishable elements, enabling us to answer report queries in linear time (with auxilliary information). Sorting the columns seems a prerequisite to an efficient strategy for inserting new features.

Figure 2 presents an interesting example. The matrix on the left can be double sorted in at least two distict ways, by permuting the rows and columns independently. In general, it is necessary to permute both rows and columns to double sort a binary matrix, although it is not immediately apparent that such a double sorting always exists. We prove existance through the following potential argument:

Theorem 14 *Every $0-1$ matrix can be double sorted by permuting rows and columns.*

The sorting algorithm induced by this potential argument can take exponential time. We provide a faster algorithm below.

Theorem 15 *An $n \times k$ binary matrix can be double sorted in $O(kn \min(n+k))$ time.*

We leave the question of whether $k \times n$ matrices can double sorted in $O(kn)$ time as an interesting open problem.

8 Acknowlegement

We thank Alex Zelikovsky introducing us to this problem.

References

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

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

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

[4] Bernard Chazelle, H. Edelsbrunner, M. Grigni, Leonidas J. Guibas, J. Hershberger, Micha Sharir, and J. Snoeyink. Ray shooting in polygons using geodesic triangulations. *Algorithmica*, 12:54–68, 1994.

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

[6] R. Cole and R. Hariharan. Dynamic LCA queries on trees. In *Proc. Tenth ACM-SIAM Symp. Discrete Algorithms (SODA)*, pages 235–244, 1999.

[7] Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin, 1997.

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

[9] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proc. 19th ACM Symp. Theory of Computing*, pages 365–371, 1987.

[10] J. Feigenbaum and S. Kannan. Dynamic graph algorithms. *Handbook of Discrete and Combinatorial Mathematics*.

[11] R. Freimer, J. Mitchell, and C. Piatko. On the complexity of shattering using arrangements. In *CCCG: Canadian Conference in Computational Geometry*, 1990.

[12] M. Garey. Optimal binary identification procedures. *SIAM J. Appl. Math.*, 23:173–186, 1972.

[13] L. Guibas and R. Sedgewick. A bichromatic framework for balanced trees. In *Proc. 19th IEEE Symp. Foundations of Computer Science*, pages 8–21, 1978.

[14] D. Gusfield. *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

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

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

[17] B. Moret. Decision trees and diagrams. *Computing Surveys*, pages 593–623, 1982.

[18] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, Cambridge, England, June 1995.

[19] 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.

[20] G. Sazaklis, E. Arkin, J. 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, 1998.

[21] 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*, pages 490–492, Nice, France, June 1997.

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

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

[24] E. Ukkonen. Constructing suffix trees on-line in linear time. In *Intern. Federation of Information Processing (IFIP '92)*, pages 484–492, 1992.

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

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