Further Results on Generalized Intersection Searching Problems: Counting, Reporting, and Dynamization

Prosenjit Gupta* Ravi Janardan* Michiel Smid†

Abstract

In a generalized intersection searching problem, a set, S, of colored geometric objects is to be preprocessed so that given some query object, q, the distinct colors of the objects intersected by q can be reported efficiently or the number of such colors can be counted efficiently. In the dynamic setting, colored objects can be inserted into or deleted from S. These problems generalize the well-studied standard intersection searching problems and are rich in applications. Unfortunately, the techniques known for the standard problems do not yield efficient solutions for the generalized problems. Moreover, previous work [JL93] on generalized problems applies only to the static reporting problems. In this paper, a uniform framework is presented to solve efficiently the counting/reporting versions of a variety of generalized intersection searching problems in static/dynamic settings. These problems include: 1-, 2-, and 3-dimensional range searching, quadrant searching, interval intersection searching, 1- and 2-dimensional point enclosure searching, and orthogonal segment intersection searching.

Keywords: Computational geometry, data structures, dynamization, intersection searching, persistence.

1 Introduction

1.1 Intersection searching problems

Problems arising in diverse areas, such as computer graphics, robotics, VLSI layout design, and databases can often be formulated as *intersection searching problems*. In a generic instance of such a problem, a set S of geometric objects is to be preprocessed into a suitable data structure

*Department of Computer Science, University of Minnesota, Minneapolis, MN 55455, U.S.A. Email: {pgupta, janardan}@cs.umn.edu. The research of these authors was supported in part by NSF grant CCR–92–00270.

†Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany. Email: michiel@mpi-sb.mpg.de. This author was supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

so that given a query object q we can answer efficiently questions regarding the intersection of q with the objects in S. The problem comes in four versions, depending on whether we want to report the intersected objects or simply count their number—the *reporting* version and the *counting* version, respectively—and whether S remains fixed or changes through insertion and deletion of objects—the *static* version and the *dynamic* version, respectively. In the dynamic version, which arises very often owing to the highly interactive nature of the above-mentioned applications, we wish to perform the updates more efficiently than simply recomputing the data structure from scratch after each update, while simultaneously maintaining fast query response times. We call these problems *standard* intersection searching problems in order to distinguish them from the *generalized* intersection searching problems that we investigate in this paper.

Typical examples of (S, q)-combinations include:

1. S is a finite set of points in $\mathcal{R}^d$ and q is a d-range: This is the *d-dimensional range searching* problem.

2. S consists of d-ranges and q is a point in $\mathcal{R}^d$: This is the *d-dimensional point enclosure searching* problem.

3. S is a finite set of points in the plane and q is a quadrant: This is the *quadrant searching* problem.

4. S consists of intervals on the real line and q is an interval: This is the *interval intersection searching* problem.

5. S consists of horizontal line segments in the plane and q is a vertical line segment: This is the *orthogonal segment intersection searching* problem.

6. S consists of line segments in the plane and q is a line segment: This is the *segment intersection searching* problem.

Due to their numerous applications, intersection searching problems have been the subject of much study and efficient algorithms have been devised for many of them. See for example [Cha86, CJ90, CJ92, EM81, EKM82, LP84, McC85, PS88, VW82], to mention just a few.

The efficiency of an intersection searching algorithm is measured by the space used by the data structure, the query time, and, in the dynamic setting, the update time. In a counting problem, these are expressed as a function of the input size n (i.e., the size of S); in a reporting problem, the space and update time are expressed as a function of n, whereas the query time is expressed as a function of both n and the output size k (i.e., the number of intersected objects) and is typically of the form $O(f(n) + k)$ or $O(f(n) + k \cdot g(n))$, for some functions f and g. Such a query time is called *output-sensitive*.

1.2 Generalized intersection searching problems

In many applications, a more general form of intersection searching arises: Here the objects in S come aggregated in disjoint groups and of interest are questions regarding the intersection of q with the groups rather than with the objects. (q intersects a group if and only if it intersects some object in the group.) In our discussion, it will be convenient to associate with each group a different color and imagine that all the objects in the group have that color. Then, in the *generalized reporting* (resp., *generalized counting*) problem, we want to report (resp., count) the distinct colors intersected by q; in the dynamic setting, an object of some (possibly new) color is inserted in S or an object in S is deleted. Note that the generalized problem reduces to the standard one when each color class has cardinality 1.

We give two examples of such generalized problems: [Databases] Consider a database of mutual funds which contains for each fund its annual total return and its beta (a real number measuring the fund's volatility). Thus each fund can be represented as a point in two dimensions. Moreover, funds are aggregated into groups according to the fund family they belong to. A typical query is to determine the families that offer funds whose total return is between, say, 15% and 20%, and whose beta is between, say, 0.9 and 1.1. This is an instance of the generalized 2-dimensional range searching problem. The output of this query enables a potential investor to initially narrow his/her search to a few families instead of having to plow through dozens of individual funds (all from the same small set of families) that meet these criteria. [VLSI layout] In the Manhattan layout of a VLSI chip, the wires (line segments) can be grouped naturally according to the circuits they belong to. A problem of interest to the designer is determining which circuits (rather than wires) become electrically connected when a new wire is added. This is an instance of the generalized orthogonal segment intersection searching problem.

1.3 Potential approaches and pitfalls

One approach to solving a generalized problem is to try to take advantage of solutions known for the corresponding standard problem. For instance, we can solve a generalized reporting problem by first determining the objects intersected by q (a standard reporting problem) and then reading off the distinct colors. However, the query time can be very high since q could intersect $\Omega(n)$ objects but only $O(1)$ distinct colors. For a generalized reporting problem, we seek query times that are sensitive to the number, i, of distinct colors intersected, typically of the form $O(f(n) + i)$ or $O(f(n) + i \cdot g(n))$, where f and g are polylogarithmic. For a generalized counting problem, the situation is worse; it is not even clear how one can extract the answer for such a problem from the answer (a mere count) to the corresponding standard problem. One could, of course, solve the corresponding reporting problem and then count the colors, but this is not efficient. Thus it is clear that different techniques are needed.

Generalized intersection searching problems were first considered in [JL93], where solutions

Generalized Problem	*Space*	*Query Time*
Interval intersection searching	n	$\log n + i$
Orthogonal segment intersection searching	$n \log n$	$\log n + i$
	n	$\log^2 n + i$
Orthogonal range searching in $\mathcal{R}^2$	$n \log^2 n$	$\log n + i$
	$n \log n$	$\log^2 n + i$
Point enclosure searching in $\mathcal{R}^d$, $d \geq 1$ is a small constant		
$d = 1$	n	$\log n + i$
$d = 2$	$n^{1.5}$	
$d = 3$	$n^{1.75}$	
$\cdots$	$\cdots$	
Segment intersection searching	$n^2 \log n$	$\log n + i$
(non-intersecting line segments)	n^2	$\log^2 n + i$

Table 1: *Summary of some previous results for generalized intersection searching problems (static reporting versions) obtained in [JL93]. All results are "big-oh" and worst-case. Here $i \geq 0$ denotes the output size, i.e., the number of distinct colors intersected. (For point enclosure searching, the exact dependence of the bounds on d can be found in [JL93].)*

using low space and efficient, output-sensitive query times were presented for several problems, including: interval intersection searching, 1- and 2-dimensional range searching, d-dimensional point enclosure searching (for constant d), and orthogonal and general segment intersection searching. (For the reader's convenience, we have summarized these results in Table 1. Additional results can be found in [JL93].) The key idea in [JL93] was to first obtain for each color class a sparse representation with the property that for certain special query objects q'—e.g., rays or grounded (i.e., 3-sided) rectangles—q' intersected the sparse representation if and only if it intersected an object of the same color, and, furthermore, q' intersected only $O(1)$ objects in the sparse representation. The objects of S were then stored suitably in a binary search tree. At each node v of the tree, a sparse representation was computed for the c-colored objects in v's subtree, for each color c. Then a standard intersection reporting structure was built on these sparse representations of different colors and stored at v. Given q, it was decomposed into two special queries q' and q'' at some node in the tree and these were used to query the standard structure at that node. The answers to q' and q'' were then combined by eliminating duplicate colors. The time for doing this elimination could be charged to the output size. (The tree representation and the decomposition of q into q' and q'' was necessary in order to guarantee the "if" part of the "if and only if" condition stated above.)

Unfortunately, the above approach does not yield efficient solutions to the counting and dynamic problems. For the counting problem, one could try to generate a sparse representation such that q' intersects *exactly* one (not just $O(1)$) objects in the representation and then use a standard counting structure instead of a reporting structure. However, care must be taken in combining the counts returned by q' and q'', since if there is a color that is included in both counts then it should

be counted only once for q; unfortunately, there seems to be no way to deduce this from the counts of q' and q''. Moreover, even if this combining were feasible, there is no "output size" to which the time for this can be charged, as was done in [JL93] for the reporting case. As for the dynamic problem, the techniques of [JL93] do not lend themselves to efficient dynamization since even a single update can result in considerable change in a sparse representation.

1.4 Summary of results and contributions

In this paper, we present efficient solutions to the counting/reporting versions of several generalized intersection searching problems in static/dynamic settings. Our results are summarized in Table 2. Note that in addition to the generalized counting problem described in Section 1.2, we also consider a second kind of counting problem, called a *type-2 counting problem*: Here we wish to report for each intersected color, the number of intersected objects of that color. (The other counting problem is implicitly a type-1 counting problem, but, for brevity, we will omit the qualifier "type-1".)

No results were known previously for any of the problems in Table 2, with the exception of the one for 2-dimensional point enclosure reporting given in Table 1. In addition to the results shown in the table, some other results obtained in the paper may be of independent interest. These include: (i) efficient solutions for generalized problems with restricted types of queries (e.g., rays, grounded rectangles) and (ii) simpler and more direct solutions to some of the static reporting problems considered in [JL93], whose bounds match those given in [JL93].

Our results for the static counting and reporting problems are based on two techniques: (i) persistent data structures and (ii) a simple geometric transformation. Roughly speaking, we use persistence as follows: To solve a given generalized problem we first identify a different but simpler generalized problem and devise a data structure for it that also supports updates (usually just insertions). We then make this structure partially persistent [DSST89] and query this persistent structure appropriately to solve the original problem. On the other hand, the transformation method works by converting a generalized problem to a different *standard* problem and solving the latter. Although this method works (at least at this time) only for the 1-dimensional range searching problem, it is crucial to our work since it also yields a dynamic data structure for the counting version of this problem, which is then used to build persistent structures for other generalized problems.

Our solutions for the dynamic problems are based on augmented data structures. Here again the transformation approach is crucial because it gives a dynamic data structure for the 1-dimensional range reporting problem, which is then used to build an augmented data structure for the dynamic quadrant reporting problem. Notice that for some dynamic problems we obtain improved solutions in the insertions-only case; these solutions constitute the building blocks of an efficient solution to the generalized 3-dimensional range reporting problem.

Thus the contribution of this paper is a uniform and general framework within which efficient

Generalized Problem	*Space*	*Query Time*	*Update Time*
1-D Range Search			
Dynamic reporting	n	$\log n + i$	$\log n$
Static counting	$n \log n$	$\log n$	
	$n \log n / \log\log n$	$\log^2 n / \log\log n$	
Dynamic counting	$n \log n$	$\log^2 n$	$\log^2 n$
Static counting (type–2)	$n \log n$	$\log n + i$	
Quadrant Search			
Static reporting	n	$\log n + i$	
Static counting	$n \log n$	$\log n$	
Dynamic reporting	$n \log n$	$\log^2 n + i \log n$	$\log^2 n$
Dyn. reptg. (insertions only)	n	$\log^2 n + i$	$\log n$ (amort.)
2-D Range Search			
Static counting	$n^2 \log^2 n$	$\log^2 n$	
Dynamic reporting	$n \log n$	$\log^2 n + i \log n$	$\log^2 n$
Dyn. reptg. (insertions only)	$n \log^2 n$	$\log^2 n + i$	$\log^3 n$ (amort.)
3-D Range Search			
Static reporting	$n \log^4 n$	$\log^2 n + i$	
Interval Intersection			
Static counting	$n \log n$	$\log n$	
Dynamic reporting	$n \log n$	$\log^2 n + i \log n$	$\log^2 n$
Dyn. reptg. (insertions only)	n	$\log^2 n + i$	$\log n$ (amort.)
1-D Point Enclosure			
Static counting	n	$\log n$	
Static counting (type–2)	n	$\log n + i$	
Dynamic reporting	n	$(i+1) \log n$	$\log n$ (amort.)
2-D Point Enclosure			
Static reporting	$n \log n$	$\log^2 n + i$	
Static counting	$n \log n$	$\log n$	
Orth. Segment Intersection			
Static counting	$n \log^2 n$	$\log^2 n$	

Table 2: *Summary of main results obtained in this paper. All bounds given are "big–oh" and, unless stated otherwise, are worst-case. Here $i \geq 0$ denotes the output size.*

solutions are derived for a wide variety of interesting, application-rich problems. This paradigm may find use in other geometric problems as well.

We note that all of our data structures can be constructed efficiently—in $O(n \cdot \text{polylog}(n))$ time (except for 2-dimensional range searching where it takes $O(n^2 \cdot \text{polylog}(n))$ time). We refrain from establishing the exact bounds in each case since this would lengthen the paper unduly and would divert attention from the main focus of the paper, which is to design solutions with low space, output-sensitive query times, and low update times.

The rest of the paper is organized as follows: Section 2 discusses some techniques that we will use frequently in the rest of the paper. Sections 3–10 contain the main results, one per section. We conclude in Section 11 with a discussion of future work.

2 Preliminaries

In this section, we review briefly persistent data structures and augmented $BB(\alpha)$ trees—two data structuring techniques that we will use frequently. We also discuss some of the issues involved in handling colors, especially in the dynamic setting. Finally, we set the notation that we use in the paper.

2.1 Persistent data structures

Ordinary data structures are *ephemeral* in the sense that once an update is performed the previous version is no longer available. In contrast, a *persistent* data structure supports operations on the most recent version as well as on previous versions. A persistent data structure is *partially persistent* if any version can be accessed but only the most recent one can be updated; it is *fully persistent* if any version can be both accessed and updated.

In [DSST89], Driscoll *et al.* describe a general technique to make persistent any ephemeral linked data structure. A *linked data structure* consists of a finite collection of nodes, each with a fixed number of fields. Each field can hold either a piece of data such as, say, an integer or a real, or a pointer to another node. The *in-degree* of a node is the number of other nodes pointing to it. Access to the structure is accomplished via one or more access pointers. Examples of linked structures include linked lists and binary search trees.

An update operation typically modifies one or more fields in the structure. We will call each modification a *memory modification*. Driscoll *et al.* showed that any linked structure whose nodes have constant in-degree can be made partially or fully persistent such that each memory modification in the ephemeral structure adds just $O(1)$ amortized space to the persistent structure and, moreover, the query time of the persistent structure is only a constant factor larger than that of the ephemeral structure.

For our purposes, partial persistence suffices. As mentioned earlier, we begin by designing

some ephemeral dynamic linked structure (e.g., a linked list or a binary search tree), which has one access pointer. Starting with an empty structure, we then perform a suitable sequence of bn updates, for some constant $b > 0$, using the technique of Driscoll *et al* and obtain a partially persistent structure, which contains all versions of the ephemeral structure. Each version has an associated "timestamp", usually an x-coordinate (or a y-coordinate) in the input. We store the access pointers of the different versions in an array, sorted by timestamp; thus any desired version can be accessed for querying by doing a binary search in the array. If $m(n)$ is the total number of memory modifications made by the update sequence, then the persistent structure uses $O(m(n))$ space.

2.2 Augmented $BB(\alpha)$ trees

$BB(\alpha)$ trees are a class of weight-balanced trees introduced by Nievergelt and Reingold [NR73]. (Here α is a positive real number which specifies the balance of the tree and for technical reasons $\alpha < 1 - 1/\sqrt{2}$.) Willard and Lueker [WL85] showed how these trees could be used to design augmented data structures with good query and update times. We review their basic approach here; for more details we refer the reader to [WL85]. At the end of this subsection, we discuss briefly the changes that are needed in order to accommodate our generalized problems.

Let P be a set of n points in d-dimensional space and suppose that we want to answer a decomposable query on P and also want to do updates on P. (A query on P is *decomposable* [SB79] if it can be answered by combining the queries on P_1 and P_2 in $O(1)$ time, where (P_1, P_2) is any partition of P.) A general approach is to construct an *augmented $BB(\alpha)$ tree* for P, which consists of a $BB(\alpha)$ tree, T, in which the points of P are stored at the leaves in sorted order according to one of the coordinates. At each internal node v of T, the points in v's subtree are stored in a suitable dynamic data structure D, which is called an *auxiliary structure* of v. A query on P is answered by identifying a suitable subset of $O(\log n)$ nodes in T, querying the D-structures at these nodes, and combining the answers. If $S(m)$ and $Q(m)$ are the space and query time of a D-structure on m points, then the augmented tree uses $O(S(n)\log n)$ space (assuming that $S(\cdot)$ grows at least linearly) and has a query time of $O(Q(n)\log n)$.

To insert/delete a point p, we search down T and insert/delete a leaf and then update the D-structure at each node on the search path. We then rebalance T via rotations. The rotations will change the set of descendant leaves of certain nodes and thus render obsolete their D-structures. These D-structures must then be rebuilt. If the rebuilding is done all at once, then, as Willard and Lueker prove, the amortized update time for the augmented tree is $O(\bar{U}(n)\log n)$, where $\bar{U}(m)$ is the amortized update time for a D-structure built on m points.

As shown in [WL85], it is possible to obtain a worst-case update time of $O(U(n)\log n)$ if a D-structure with a worst-case update time of $U(m)$ is available, where m is the number of points in the D-structure. The idea is to spread the rebuilding work at a node v over a sequence of future

updates in v's subtree. (Thus an update operation will perform some rebuilding work at each node on the search path.) While v's D-structure is under construction, queries at v are answered by querying its children and combining the answers. This requires that the structures at v's children be up-to-date and, moreover, the children should not undergo rotation while v's structure is under construction. Willard and Lueker show how to assure this by requiring that a certain condition be satisfied for a node to be eligible for rotation. Moreover, they show that it is sufficient to perform just $O(1)$ rebuilding operations at v and still guarantee that v's structure will be ready before v becomes eligible for rotation. Occasionally, some node may become highly unbalanced before it is eligible for a rebalancing rotation. Fortunately, as shown in [WL85], the subtree of such a node will have size just $O(1)$ and so the node can be handled by simply rebuilding its entire subtree into a balanced subtree.

We will use augmented $BB(\alpha)$ trees in several places—specifically in Sections 3.1.3, 4.2, 5.2, and 8.3. In Section 3.1.3, we use these trees to solve a standard range searching problem and so the above discussion applies unchanged. In Sections 4.2 and 5.2, we use these trees to solve generalized range searching problems, while in Section 8.3 we use these trees to solve the dynamic generalized point enclosure problem on intervals. Here the main difference from the above discussion is in the querying: As before, when we have to do a query at a node v whose structures are under construction, we query v's children and combine the answers. However, combining the answers takes more work since duplicate colors must be eliminated. This can be done in $O(i)$ time (not $O(1)$ time) by using a technique to be described in Section 2.3.

2.3 Issues in the handling of colors

We discuss some of the issues that arise in the encoding and handling of colors, especially in the dynamic setting. Throughout the paper, we will assume that our algorithms incorporate the mechanisms that we describe here and so we will not repeat them afterwards.

The number of colors for a given problem can range from 1 to n. We encode each color as an integer in the range $[1, n]$. Among other things, this allows us to impose a total order on the colors and also allows us to use colors as array indices.

In many of our generalized reporting problems, when answering a query we may encounter the same color more than once (but no more than $O(1)$ times). Our goal is to eliminate the duplicate colors efficiently before we output the answer. This situation arises in several different ways, such as, for instance: (i) In augmented $BB(\alpha)$ trees, when we answer a query at a node whose auxiliary structure is under construction by querying its two children (see Section 2.2), (ii) when answering a query by decomposing the query object into $O(1)$ simpler objects and querying with the latter, (iii) when querying with a single query object which intersects $O(1)$ input objects of the same color, and (iv) combinations of the above three cases.

We can eliminate duplicate colors by using an array, $C[1 : n]$, of colors to keep track of the

distinct colors that are found during a query. We also store the distinct colors found in a linked list. After the query, C can be reset in time proportional to the output size by scanning the list. Thus the asymptotic query time is unaffected.

While this approach works fine in the static case, it can lead to problems in the dynamic setting. Assume that n is the current size of S and let C have size n. When inserting an object of a new color, we encode the color with an unassigned integer; when deleting the last object of some color from S, we return the integer encoding the color to the pool of unassigned integers. Now, if many new colors are inserted in S, then we could run out of space in C; conversely, if many colors are deleted from S, then the size of C could become much larger than the new size of S.

To overcome this problem, we periodically build a new version of the entire data structure and during this process we re-encode the distinct colors in S into a larger or smaller range of integers, as appropriate, and also create a new C. While the rebuilding is underway, the current structure continues to be in use for queries and updates. The rebuilding is done by piggy-backing a small amount of rebuilding work onto each future update on the current structure. The key is to make sure that the rebuilding can be done without affecting the update time of the current structure and that the new structure can take over from the old one in time. (This idea is reminiscent of the global rebuilding technique of Overmars [Ove83].)

Specifically, let n_0 have been the size of S when the previous rebuilding was initiated (i.e., the rebuilding that resulted in the current structure) and assume that C was created with size $2n_0$. When the number of updates (insertions and deletions) on the current structure exceeds, say, $n_0/3$, we start building a new structure, with the goal of having it take over by the time an additional $n_0/3$ updates have been done on the current structure. If n_1 is the size of S at this point, i.e., after $n_0/3$ updates, then the new C has size $2n_1$. Note that n_1 satisfies $2n_0/3 \leq n_1 \leq 4n_0/3$. With each future update, we do four rebuilding insertions on the new structure and also do that update on it. Since $n_1 \leq 4n_0/3$, it is clear that the new structure will be ready within the next $n_0/3$ updates.

Note that at any time during the rebuilding the size of S is less than $n_1 + n_0/3 \leq 5n_0/3$ and greater than $n_1 - n_0/3 \geq n_0/3$. Thus, we will neither run out of space in C, which has size $2n_0$, nor will C's size become too large compared to the size of S. Clearly, the query time is unaffected asymptotically. Let $I(m)$ and $D(m)$ denote, respectively, the insertion and deletion time for our data structure when it has m points, exclusive of the rebuilding insertions. Then, with the rebuilding insertions, the insertion time of the current structure is $O(I(n_0) + I(n_0)) = O(I(n_0))$ and the deletion time is $O(D(n_0) + I(n_0))$. In all our algorithms, $I(m)$ and $D(m)$ are of the same order and so the overall deletion time is $O(D(n_0))$.

2.4 Notation

Finally, for ease of reference, we summarize some of our notation and terminology. As mentioned before, we use n to denote the input size (e.g., the number of points/line segments/rectangles etc.

in S); in the dynamic setting, n stands for the *current* size of S. We use i to denote the output size of a generalized reporting or type-2 counting problem, i.e., the number of distinct colors intersected.

If v is a node in some binary tree T, then $T(v)$ denotes the subtree rooted at v and $left(v)$ and $right(v)$ denote the left and right child of v, respectively. If a set of points is stored in nondecreasing order (according to some coordinate) from left to right at the leaves of T, then the *range* of v is the closed interval bounded by the leftmost and rightmost points in $T(v)$.

Unless stated otherwise, all bounds in the paper are worst-case.

3 Generalized 1-dimensional range searching

Let S be a set of n colored points on the x-axis. We show how to preprocess S so that for any query interval, q, we can solve efficiently the dynamic reporting problem, the static and dynamic counting problems, and the static type-2 counting problem, as stated in Table 2.

Previously, the static reporting version of the problem was considered in [JL93] and an $O(n)$-space and $O(\log n + i)$-query time algorithm was given. Using a result of [Lop91], the dynamic version of the problem can be solved in $O(n \log n)$ (resp., $O(n)$) space, $O(\log n + i)$ (resp., $O(\log^2 n + i)$) query time, and $O(\log^2 n)$ (resp., $O(\log n)$) amortized update time. Our dynamic algorithm is more efficient and much simpler. Moreover, the static version of this structure is even simpler and solves the static problem within the same bounds as [JL93] but, unlike [JL93], does not use fractional cascading. No results were known before for any of the counting problems.

Our schemes for the reporting/counting problems are based on a simple transformation, which converts the original generalized reporting/counting problem to an instance of some standard reporting/counting problem, for which efficient solutions are known. Subsequently, we also present an alternative solution, with the same performance bounds, for the static counting problem and a solution to the static type-2 counting problem. This approach is based on persistent data structures and has the advantage of being generalizable to other problems as well.

3.1 A transformation-based approach

We first describe the transformation and prove its correctness.

For each color c, we sort the distinct points of that color by increasing x-coordinate. For each point p of color c, let $pred(p)$ be its predecessor in the sorted order; for the leftmost point of color c, we take the predecessor to be the point $-\infty$. We then map p to the point $p' = (p, pred(p))$ in the plane and associate with it the color c. Let S' be the resulting set of points. Given a query interval $q = [l, r]$, we map it to the grounded rectangle $q' = [l, r] \times (-\infty, l)$.

Lemma 3.1 *There is a point of color c in $q = [l, r]$ if and only if there is a point of color c in $q' = [l, r] \times (-\infty, l)$. Moreover, if there is a point of color c in q', then this point is unique.*

Proof ($\Leftarrow$) Let p' be a c-colored point in q', where $p' = (p, pred(p))$ for some c-colored point $p \in S$. Since p' is in $[l, r] \times (-\infty, l)$, it is clear that $l \leq p \leq r$ and so $p \in [l, r]$.

($\Rightarrow$) Let p be the leftmost point of color c in $[l, r]$. Thus $l \leq p \leq r$ and since $pred(p) \notin [l, r]$, we have $l > pred(p)$. It follows that $p' = (p, pred(p))$ is in $[l, r] \times (-\infty, l)$. We prove that p' is the only point of color c in q'. Suppose for a contradiction that $t' = (t, pred(t))$ is another point of color c in q'. Thus we have $l \leq t \leq r$. Since $t > p$, we also have $pred(t) \geq p \geq l$. Thus $t' = (t, pred(t))$ cannot lie in q'—a contradiction. The claim follows. $\square$

Lemma 3.1 implies that we can solve the generalized 1-dimensional range reporting (resp., counting) problem by simply reporting the points in q' (resp., counting the number of points in q'), without regard to colors. In other words, we have reduced the generalized reporting (resp., counting) problem to the standard grounded range reporting (resp., counting) problem in two dimensions. In the dynamic case, we also need to update S' when S is updated. We discuss these matters in more detail below.

3.1.1 The dynamic reporting problem

Our data structure consists of the following: For each color c, we maintain a balanced binary search tree, T_c, in which the c-colored points of S are stored in increasing x-order. We maintain the colors themselves in a balanced search tree, CT, and store with each color c in CT a pointer to T_c. We also store the points of S' in a *balanced priority search tree* (PST) [McC85]. Recall that a PST on m points occupies $O(m)$ space, supports insertions and deletions in $O(\log m)$ time, and can be used to report the k points lying inside a grounded query rectangle in $O(\log m + k)$ time via the query *Enumerate_Rectangle* in [McC85]. (Although this query is designed for query ranges of the form $[l, r] \times (-\infty, l]$, it can be trivially modified to ignore the points on the upper edge of the range, without affecting its performance.) Clearly, the space used by the entire data structure is $O(n)$.

To answer a query $q = [l, r]$, we simply query the PST with $q' = [l, r] \times (-\infty, l)$ and report the colors of the points found. Correctness follows from Lemma 3.1. The query time is $O(\log n + k)$, where k is the number of points inside q'. By Lemma 3.1, $k = i$, and so the query time is $O(\log n + i)$.

Suppose that a c-colored point p is to be inserted in S. If $c \notin CT$, then we create a tree T_c containing p, insert $p' = (p, -\infty)$ into the PST, and insert c, with a pointer to T_c, in CT. Suppose that $c \in CT$. Let u be the successor of p in T_c. If u exists, then we set $pred(p)$ to $pred(u)$ and $pred(u)$ to p; otherwise, we set $pred(p)$ to the rightmost point in T_c. We then insert p into T_c, $p' = (p, pred(p))$ into the PST, delete the old u' from the PST, and insert the new u' into it.

Deletion of a point p of color c is essentially the reverse. We delete p from T_c. Then we delete p' from the PST and if p had a successor, u, in T_c then we reset $pred(u)$ to $pred(p)$, delete the old u' from the PST, and insert the new one. If T_c becomes empty in the process, then we delete c from CT. Clearly, the update operations are correct and take $O(\log n)$ time.

Theorem 3.1 *Let S be a set of n colored points on the real line. S can be preprocessed into a data structure of size $O(n)$ such that the i distinct colors of the points of S that are intersected by any query interval can be reported in $O(\log n + i)$ time and points can be inserted and deleted online in S in $O(\log n)$ time.* □

For the static reporting problem, we can dispense with CT and the T_c's and simply use a static form of the PST to answer queries. This provides a simple $O(n)$-space, $O(\log n + i)$-query time alternative to the solution given in [JL93].

3.1.2 The static counting problem

We store the points of S' in nondecreasing x-order at the leaves of a balanced binary search tree, T, and store at each internal node t of T an array A_t containing the points in t's subtree in nondecreasing y-order. The total space is clearly $O(n \log n)$. To answer a query, we determine $O(\log n)$ canonical nodes v in T such that $[l, r]$ covers v's range but not the range of v's parent. Using binary search we determine in each canonical node's array the highest array position containing an entry less than l (and thus the number of points in that node's subtree that lie in q') and add up the positions thus found at all canonical nodes. The correctness of this algorithm follows from Lemma 3.1. The total query time is $O(\log^2 n)$.

We can reduce the query time to $O(\log n)$ as follows: At each node t we create a linked list, B_t, which contains the same elements as A_t and maintain a pointer from each entry of B_t to the same entry in A_t. We then apply the technique of fractional cascading [CG86] to the B-lists, so that after an initial $O(\log n)$-time binary search in the B-list of the highest canonical node, the correct position in the B-list of each of the other canonical nodes can be found directly in $O(1)$ time. (To facilitate binary search in a B-list, we build a balanced search tree on each B-list after the fractional cascading step.) Once the position in a B-list is known, the appropriate position in the corresponding A-array can be found in $O(1)$ time.

We now give an alternative solution which reduces the space to $O(n \log n / \log \log n)$ at the expense of an $O(\log^2 n / \log \log n)$ query time. We sort the points of S' in nondecreasing y-order, breaking ties arbitrarily. We then draw a horizontal line after every $\lceil n/\alpha \rceil$ points in this sorted list, where $\alpha \leq n$ is a parameter to be fixed later. This partitions the plane into at most α horizontal strips, each of size $\lceil n/\alpha \rceil$ (with the possible exception of one strip which may contain fewer points). With each strip, we store an array containing its points in nondecreasing x-order, ties broken arbitrarily. We perform this processing recursively within each strip, stopping when a strip has size 1. The number of levels is $O(\log n / \log \alpha)$ and each level uses $O(n)$ space. Thus the total space is $O(n \log n / \log \alpha)$.

A query is answered as follows: We start at the topmost level of the partition. Note that q' will completely span, in the y direction, all but possibly one strip at this level. For each spanned strip, we can determine the number of points of the strip lying in q' from the positions of l and r in the

strip's array. This takes $O(\log(n/\alpha))$ time, via binary search, and so the total time for all spanned strips is $O(\alpha \cdot \log(n/\alpha))$. Using fractional cascading this can be reduced to $O(\alpha + \log(n/\alpha))$. The one partially-spanned strip is handled recursively since q' behaves like a grounded rectangle for this strip. This yields a total query time of $O((\alpha + \log(n/\alpha)) \log n / \log \alpha)$. Choosing $\alpha = \lceil \log n \rceil$ gives a query time of $O(\log^2 n / \log\log n)$ and a space bound of $O(n \log n / \log\log n)$.

Theorem 3.2 *Let S be a set of n colored points on the real line. S can be preprocessed into a data structure of size $O(n \log n)$ (resp., $O(n \log n / \log\log n)$) such that the number of distinctly-colored points of S that are intersected by any query interval can be determined in $O(\log n)$ (resp., $O(\log^2 n / \log\log n)$) time.* $\square$

3.1.3 The dynamic counting problem

We store the points of S' using the same basic two-level tree structure as in Section 3.1.2. However, T is now a $BB(\alpha)$ tree (Section 2.2) and the auxiliary structure, $D(t)$, at each node t of T is a balanced binary search tree where the points are stored at the leaves in left to right order by nondecreasing y-coordinate. To facilitate the querying, each node v of $D(t)$ stores a count of the number of points in its subtree. Given a real number, r, we can determine in $O(\log n)$ time the number of points in $D(t)$ that have y-coordinate less than r by searching for r in $D(t)$ and adding up the count for each node of $D(t)$ that is not on the search path but is the left child of a node on the path. It should be clear that $D(t)$ can be maintained in $O(\log n)$ time under updates.

In addition to the two-level structure, we also use the trees T_c and the tree CT, described in Section 3.1.1, to maintain the correspondence between S and S'. We omit further discussion about the dynamic maintenance of the T_c's and of CT.

Queries are answered as in the static case, except that at each auxiliary structure we use the above-mentioned method to determine the number of points with y-coordinate less than l. Thus the query time is $O(\log^2 n)$. (We cannot use fractional cascading here.)

Insertion/deletion of a point is done using the worst-case updating strategy described in Section 2.2. Since here we have $U(n) = O(\log n)$ for an auxiliary structure, the worst-case update time for the entire data structure is $O(\log^2 n)$.

Theorem 3.3 *Let S be a set of n colored points on the real line. S can be preprocessed into a data structure of size $O(n \log n)$ such that the number of distinctly-colored points of S that are intersected by any query interval can be determined in $O(\log^2 n)$ time and points can be inserted and deleted online in S in $O(\log^2 n)$ worst-case time.* $\square$

3.2 A persistence-based approach

3.2.1 The static counting problem

In this subsection, we give an alternative scheme for the static counting problem which also achieves $O(n \log n)$ space and $O(\log n)$ query time. We first solve a related problem, namely when q is of the form $[a, \infty)$ and where we also wish to insert and delete points.

For each color c, we determine the maximal (i.e., rightmost) point of that color and store all these maximal points at the leaves of a balanced binary search tree T. At each node of T, we store a count of the number of leaves in the node's subtree. Given q, we identify $O(\log n)$ canonical nodes v such that q spans the range of v but not the range of v's parent and sum the counts at all such nodes v.

To support updates we also maintain a balanced search tree, CT, which stores all colors. With each color c in CT, we store a pointer to a balanced binary search tree T_c for all c-colored points. To insert a point p of, say, color c, we first search in CT for c. If c does not exist in CT, then we make a binary tree T_c consisting of p alone and store c, with a pointer to T_c, in CT. If c exists in CT, then we insert p in T_c and if p is to the right of the current maximal point of color c, then we delete the latter from T and insert p as the new maximal point of color c. Clearly, the insertion time is $O(\log n)$ and the number of memory modifications is $O(\log n)$. (Although we do not require deletion capability, we note that deletions can also be done within the above bounds.)

Lemma 3.2 *A set S of n colored points on the real line can be preprocessed into a data structure of size $O(n)$ such that the number of distinctly-colored points contained in any query interval $q = [a, \infty)$ can be reported in $O(\log n)$ time. The structure supports insertions and deletions in $O(\log n)$ time with $O(\log n)$ memory modifications.* □

We can now solve the problem for a query interval $q = [a, b]$ as follows: Using the approach of Driscoll *et al.* [DSST89] described in Section 2.1, we build a partially persistent version of the data structure of Lemma 3.2, by treating the x-coordinate as time and inserting the points by nondecreasing x-coordinate (ties broken arbitrarily) into an initially-empty data structure. (Note that only the tree T needs to be made persistent. CT and the trees T_c are only needed to do updates efficiently in the current version. They are not needed for queries and can, in fact, be discarded once the persistent version of T has been built.) Given $q = [a, b]$, we determine the version corresponding to the greatest x-coordinate that is less than or equal to b and then query that version with the interval $[a, \infty)$.

Theorem 3.4 *A set S of n colored points on the real line can be preprocessed into a data structure of size $O(n \log n)$ such that the number of distinctly-colored points contained in any query interval $q = [a, b]$ can be reported in $O(\log n)$ time.*

Proof First note that the structure of Lemma 3.2 has constant in-degree and so the results of [DSST89] (Section 2.1) apply. The correctness of the query algorithm follows from the observation that since the version accessed does not contain any point to the right of b, querying it with $q = [a, b]$ is equivalent to querying it with $q = [a, \infty)$. The query time follows from Lemma 3.2. To build the structure we do n insertions, each of which causes $O(\log n)$ memory modifications. Thus, the persistent structure uses $O(n \log n)$ space. □

3.2.2 The static type-2 counting problem

We wish to preprocess a set S of n colored points on the x-axis, so that for each color intersected by a query interval $q = [a, b]$, the number of points of that color in q can be reported efficiently.

We first show how to solve the problem for $q = [a, \infty)$. We sort the points of S in non-decreasing order (ties broken arbitrarily) as $p_1, p_2, \ldots, p_n$, and store them and the points $p_0 = -\infty$ and $p_{n+1} = \infty$ in an array A. This defines $n + 1$ intervals, some possibly empty. Let I_j denote interval $(p_j, p_{j+1}]$. Notice that for any position of a in I_j, the answer is invariant. With I_j we store a list L_j: Let m be the number of points of color c with x-coordinate greater than or equal to p_{j+1}. If $m > 0$, then L_j contains the entry $< c, m >$. (The entries in L_j appear in no particular order.) To answer a query, the interval I_j containing a is located by binary search in A. Then L_j is output in $\Theta(i)$ time. The total query time is $O(\log n + i)$ and the space is $O(n^2)$.

The space can be reduced to $O(n)$ by observing that L_{j-1} and L_j are nearly the same; the only difference is that if p_j is of color c and the entry $< c, m >$ is stored in L_{j-1}, then, if $m > 1$, then the entry $< c, m-1 >$ is stored in L_j and, if $m = 1$, then there is no entry for c in L_j. Treating the x-coordinate as time, we store all the L-lists in a partially persistent doubly-linked list as follows: We start at p_0 with the doubly-linked list L_0 which contains an entry $< c, m >$ for each distinct color c. For $j = 1, 2, \ldots, n$, if p_j is of color c, then we obtain L_j by performing a decrement or a delete operation on the entry for c in L_{j-1}. Each operation takes $O(1)$ time and causes $O(1)$ memory modifications. (The entry for c in L_{j-1} can be accessed in $O(1)$ time by maintaining an array which stores for each color c a pointer to the entry for c in the most recent list.) Since there are n operations in all, the entire structure occupies $O(n)$ space. To answer a query, we access the appropriate version by binary search in A. By the results in [DSST89], the time to output the list is still $\Theta(i)$.

Lemma 3.3 *A set S of n colored points on the real line can be preprocessed into a data structure of size $O(n)$ such that for any query interval $[a, \infty)$, a type-2 counting query can be answered in $O(\log n + i)$ time, where i is the output size.* □

Using the above structure, we can handle a finite query interval $[a, b]$ as follows: We associate the root r of a balanced binary search tree T with the point(s) of S with median x-coordinate. At

r we store three structures, *Left*(r), *Right*(r), and *Mid*(r): *Left*(r) is an instance of the structure of Lemma 3.3, built on the points to the left of the median; *Right*(r) is a symmetric structure (for queries of the form $(-\infty, b]$) built on the points to the right of the median; and *Mid*(r) is a linked list containing entries of the form $< c, m >$ for each color c, if there are $m > 0$ points of color c at the median x-coordinate. We also store the median x-coordinate in a field $X(r)$. The left and right subtrees of r are built recursively on the points to the left and to the right of the median, respectively.

A query is answered by searching down T and locating the highest node v such that $a \leq X(v) \leq b$. If v does not exist, then we stop. Otherwise, we query *Left*(v) with $[a, \infty)$ and *Right*(v) with $(-\infty, b]$. We take the two lists of answers output and the list *Mid*(v) and if a color appears in more than one of these lists, then we add up its counts before outputting the final answer.

Theorem 3.5 *A set S of n colored points on the real line can be preprocessed into a data structure of size $O(n \log n)$ such that for any query interval $q = [a, b]$, a type-2 counting query can be answered in $O(\log n + i)$ time, where i is the output size.* □

4 Generalized quadrant searching

Let S be a set of n colored points in the plane. For any query point $q = (a, b)$, the *northeast quadrant* of q, denoted by $NE(q)$, is the set of all points (x, y) in the plane such that $x \geq a$ and $y \geq b$. We show how to preprocess S so that for any $NE(q)$ we can solve efficiently the static reporting and counting problems and the dynamic and semi-dynamic (i.e., insertions-only) reporting problems, as stated in Table 2. We present some applications of these results in Section 5.2 and Section 7.

4.1 The static reporting and counting problems

Consider the reporting problem. We first solve a related problem, namely: preprocess a set of colored points on the real line so that the i distinct colors intersected by a query interval $[a, \infty)$ can be reported efficiently and, moreover, points can be inserted efficiently.

For each color, we determine the rightmost point of that color and store all these points in sorted order at the leaves of a balanced search tree T and thread the leaves into a doubly-linked list L. Moreover, we store all the colors in a balanced search tree CT. With each color, we store the rightmost point of that color. A query can be answered by simply scanning L from right to left, stopping when a point to the left of a is encountered. The query time is $\Theta(i)$ and the space is $O(n)$. To insert a point p of color c we use CT to determine if p is to the right of the c-colored point (if any) currently in L and if so then we delete that point from T and insert p. This takes $O(\log n)$ time and causes $O(1)$ memory modifications in the list L.

To solve the quadrant reporting problem, we consider the points of S in non-increasing y-order and insert their x-coordinates into a partially persistent version of the above list L. (The trees T

and CT are needed only to do updates efficiently in the current list and are not used for queries. Therefore, we need not make them persistent.) Given $NE(q)$, where $q = (a, b)$, we determine the smallest y-coordinate in S that is greater than or equal to b and query the corresponding version of L with $[a, \infty)$.

Theorem 4.1 *A set S of n colored points in the plane can be preprocessed into a data structure of size $O(n)$ such that for any query point q, the i distinct colors of the points lying in the northeast quadrant of q can be reported in $O(\log n + i)$ time.*

Proof To see that the method is correct, note that only those points that are at least as high as q can possibly be in $NE(q)$. The version accessed consists of precisely these points. Moreover, since $NE(q)$ is infinite upwards, any such point (x, y) is in $NE(q)$ if and only if x is in $[a, \infty)$. The query time is $O(\log n + i)$ since $O(\log n)$ time is required to access the correct version and $\Theta(i)$ additional time is required to query it. There are n insertions into L, each causing $O(1)$ memory modifications. Thus, the space is $O(n)$. □

For the static quadrant counting problem, we use a similar approach. However, the 1-dimensional problem that we need to solve now is counting the number of distinct colors in any query interval $[a, \infty)$. For this we use the structure of Lemma 3.2. To solve the quadrant problem, we make this structure partially persistent.

Theorem 4.2 *A set S of n colored points in the plane can be preprocessed into a data structure of size $O(n \log n)$ such that the number of distinctly-colored points lying in the northeast quadrant of any query point q can be reported in $O(\log n)$ time.* □

4.2 Dynamic quadrant reporting

We can solve the dynamic problem using the well-known technique of range restriction [WL85]. We store the points of S in nondecreasing y-order from left to right at the leaves of a $BB(\alpha)$ tree T. At each node v of T, we store an auxiliary structure, $D(v)$, for dynamic generalized 1-dimensional range searching (Theorem 3.1). $D(v)$ is built on the x-coordinates of the points in $T(v)$. Given a query $NE(q)$, where $q = (a, b)$, we search in T with the interval $[b, \infty)$ and identify a set V of $O(\log n)$ canonical nodes such that for each $v \in V$, $[b, \infty)$ covers the range of v but not the range of v's parent. At each such v, we query $D(v)$ with the interval $[a, \infty)$. The answer to the quadrant query then is a list of the distinct colors output at the nodes of V. Insertion/deletion of a point is done using the worst-case updating strategy described in Section 2.2. (In particular, note the discussion at the end of Section 2.2 on how queries are performed at a node whose auxiliary structure is under construction.) The correctness of the method and the space and query time bounds are easy to establish. By Theorem 3.1, any D-structure has an update time of $U(n) = O(\log n)$ and so the overall structure has update time $O(U(n) \log n) = O(\log^2 n)$.

Theorem 4.3 *Let S be a set of n colored points in the plane. S can be stored in a data structure of size $O(n \log n)$ such that for any query point q, the i distinct colors of the points lying in q's northeast quadrant can be reported in $O(\log^2 n + i \log n)$ time and points can be inserted and deleted in $O(\log^2 n)$ time.* $\square$

4.3 Dynamic quadrant reporting: the insertions-only case

In this subsection, we show that if only insertions are allowed then the results of Section 4.2 can be improved substantially by using a different approach. The scheme uses $O(n)$ space, has a query time of $O(\log^2 n + i)$, and an amortized insertion time of $O(\log n)$. More importantly, this result is one of the building blocks of an efficient scheme for generalized 3-dimensional range searching which we describe in Section 6.

For each color c, we determine the c-maximal points. (A point p is called *c-maximal* if it has color c and if there are no points of color c in p's northeast quadrant.) We discard all points of color c that are not c-maximal. In the resulting set, let the predecessor, *pred*(p), of a c-colored point p be the c-colored point that lies immediately to the left of p. (For the leftmost point of color c, the predecessor is the point $(-\infty, \infty)$.) With each point $p = (a, b)$, we associate the horizontal segment with endpoints (a', b) and (a, b), where a' is the x-coordinate of *pred*(p). This segment gets the same color as p. Let S_c be the set of such segments of color c. (See Figure 1, where the solid segments belong to S_c. Note that these segments form a staircase.) The data structure consists of the following:

1. A structure $\mathcal{T}$ storing the segments in the sets S_c, where c runs over all colors. $\mathcal{T}$ supports the following query: given a point q in the plane, report the segments that are intersected by the upward-vertical ray starting at q. Moreover, it allows segments to be inserted and deleted. We implement $\mathcal{T}$ as the structure given in [CJ90]. This structure uses $O(n)$ space, supports insertions and deletions in $O(\log n)$ time, and has a query time of $O(\log^2 n + l)$, where l is the number of segments intersected.

2. A balanced search tree, CT, storing all colors. For each color c, we maintain a balanced search tree, T_c, storing the segments of S_c by increasing y-coordinate. This structure allows us to dynamically maintain S_c when a new c-colored point p is inserted. The approach is as follows: By doing a binary search in T_c we can determine whether or not p is c-maximal in the current set of c-maximal points, i.e., the set of right endpoints of the segments of S_c. If p is not c-maximal, then we simply discard it. If p is c-maximal, then let $s_1, \ldots, s_k$ be the segments of S_c whose right endpoints are in the southwest quadrant of p. (See Figure 1.) We do the following: (i) delete $s_1, \ldots, s_k$ from T_c; (ii) insert into T_c the horizontal segment which starts at p and extends leftwards upto the x-coordinate of the c-maximal point above p; and (iii) truncate the segment of S_c that is below p by keeping only the part of it that extends

leftwards upto p's x-coordinate. The entire operation can be done in $O((k+1)\log n)$ time (or even $O(\log n + k)$ time).

Note that $s_1, \ldots, s_k$ do not exist if p is above the highest c-maximal point or below the lowest c-maximal point and also may not exist for some intermediate positions of p. Moreover, if p is above the highest c-maximal point then in step (ii) we insert a horizontal, leftward-directed ray which starts at p (rather than a finite segment). If p is below the lowest c-maximal point, then step (iii) need not be done.

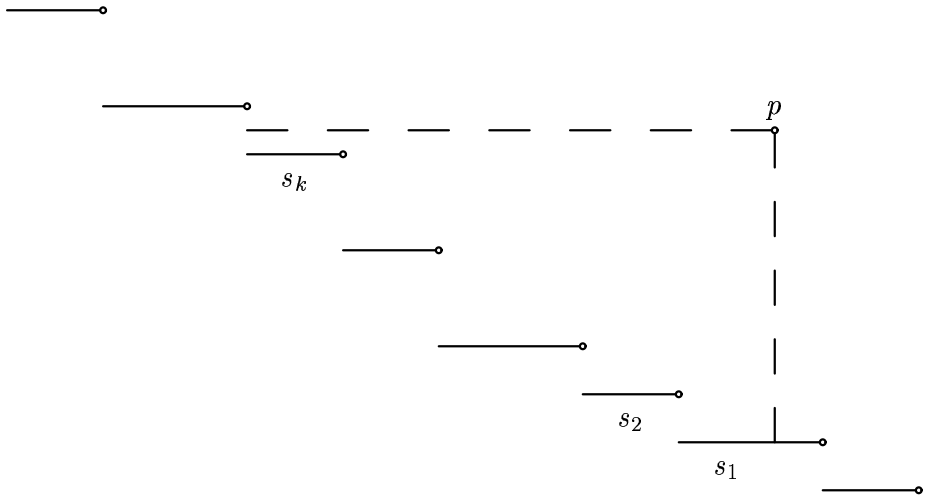

Figure 1: *Insertion of a c-colored point p into the data structure for generalized semi-dynamic quadrant reporting.*

Let us now consider how to answer a quadrant query, $NE(q)$, and how to insert a point in S. To answer $NE(q)$, we query $\mathcal{T}$ with the upward-vertical ray from q and report the colors of the segments intersected. The correctness of this algorithm follows from the easily proved facts that (i) a c-colored point lies in $NE(q)$ if and only if a c-maximal point lies in $NE(q)$ and (ii) if a c-maximal point is in $NE(q)$, then the upward-vertical ray from q must intersect a segment of S_c. The correctness of $\mathcal{T}$ guarantees that only the segments intersected by this ray are reported. Since the query can intersect at most two segments in any S_c, we have $l \leq 2i$, and so the query time is $O(\log^2 n + i)$.

Let p be a c-colored point that is to be inserted into S. If c is not in CT, then we insert it into CT and insert the horizontal, leftward-directed ray emanating from p into a new structure T_c. If c is present already, then we update T_c as just described. In both cases, we then perform the same updates on $\mathcal{T}$. Hence, an insertion takes $O((k+1)\log n)$ time.

What is the total time for doing n insertions into an initially-empty set? For each insertion, we can charge the $O(\log n)$ time to delete a segment s_i, $1 \leq i \leq k$, to s_i itself. Notice that none of these segments will reappear. Thus each segment is charged at most once. Moreover, each of

these segments has some previously inserted point as a right endpoint. It follows that the number of segments existing over the entire sequence of insertions is $O(n)$ and so the total charge to them is $O(n \log n)$. The rest of the cost for each insertion ($O(\log n)$ for the binary search plus $O(1)$ for steps (ii) and (iii)) we charge to p itself. Since any p is charged in this mode only once, the total charge incurred in this mode by all the inserted points is $O(n \log n)$. Thus the time for n insertions is $O(n \log n)$, which implies an amortized insertion time of $O(\log n)$.

Theorem 4.4 *Let S be a set of n colored points in the plane. There exists a data structure of size $O(n)$ such that for any query point q, we can report the i distinct colors of the points that are contained in the northeast quadrant of q in $O(\log^2 n + i)$ time. Moreover, if we do n insertions into an initially-empty set then the amortized insertion time is $O(\log n)$.* $\square$

5 Generalized 2-dimensional range searching

We show how to preprocess a set S of n colored points in the plane so that for any axes-parallel query rectangle $q = [a, b] \times [c, d]$, we can solve efficiently the static counting problem and the dynamic and semi-dynamic reporting problems, as stated in Table 2. No results were known for these problems other than the ones given in Table 1 for the static reporting version.

5.1 The static counting problem

We first solve the problem for $q' = [a, \infty) \times [c, d]$. We sweep over the points of S by nonincreasing x-coordinate and insert their y-coordinates into a partially persistent version of the structure of Theorem 3.3 for dynamic 1-dimensional range counting. Given q', we access the version corresponding to the smallest x-coordinate x_0 such that $x_0 \geq a$ and query it with $[c, d]$.

To solve the problem for $q = [a, b] \times [c, d]$, we build for each distinct x-coordinate $\hat{x}$ in S an instance of the above structure for those points whose x-coordinate is less than or equal to $\hat{x}$. Given q, we access the structure corresponding to the greatest x-coordinate x_1 in S such that $x_1 \leq b$ and query it with q'.

Theorem 5.1 *A set S of n colored points in the plane can be preprocessed into a data structure of size $O(n^2 \log^2 n)$ such that for any axes-parallel query rectangle $q = [a, b] \times [c, d]$, the number of distinctly-colored points in q can be reported in $O(\log^2 n)$ time.*

Proof Consider the structure for q'. The correctness of the query algorithm follows from the observations that only those points of S with x-coordinate greater than or equal to x_0 can possibly be in q' and the version accessed contains just these points. Moreover, since q' is infinite to the right of a, the problem becomes 1-dimensional in y. Since the structure of Theorem 3.3 supports

insertions in $O(\log^2 n)$ time, the number of memory modifications per update is also $O(\log^2 n)$. Thus the persistent structure uses $O(n \log^2 n)$ space. The query time follows from Theorem 3.3.

The correctness of the structure for q can be established similarly. The space and query time follow from the bounds for q'. $\Box$

5.2 The dynamic and semi-dynamic reporting problems

We can solve the dynamic problem using the approach for dynamic quadrant reporting in Section 4.2. The only difference is that we search in T using $[c, d]$ to identify the canonical nodes and query at each such node with $[a, b]$. This immediately gives:

Theorem 5.2 *A set S of n colored points in the plane can be preprocessed into a structure of size $O(n \log n)$ such that the i distinct colors of the points lying inside an axes-parallel query rectangle can be reported in $O(\log^2 n + i \log n)$ time. The structure supports updates in $O(\log^2 n)$ time.* $\Box$

We can improve upon Theorem 5.2 in the insertions-only case by using a different approach. We will use this improved result in Section 6. Our solution is based on the semi-dynamic quadrant reporting structure of Section 4.3. We first show how to solve the problem for $q' = [a, b] \times [c, \infty)$. We store the points of S in sorted order by x-coordinate at the leaves of a $BB(\alpha)$ tree T'. At each internal node v, we store an instance of the structure of Theorem 4.4 for *NE*-queries (resp., *NW*-queries) built on the points in v's left (resp., right) subtree. Let $X(v)$ denote the average of the x-coordinate in the rightmost leaf in v's left subtree and the x-coordinate in the leftmost leaf of v's right subtree; for a leaf v, we take $X(v)$ to be the x-coordinate of the point stored at v.

To answer a query q', we do a binary search down T', using $[a, b]$, until either the search runs off T' or a (highest) node v is reached such that $[a, b]$ intersects $X(v)$. In the former case, we stop. In the latter case, if v is a leaf, then if v's point is in q' we report its color. If v is a non-leaf, then we query the structures at v using the *NE*-quadrant and the *NW*-quadrant derived from q' (i.e., the quadrants with corners at (a, c) and (b, c), respectively), and then combine the answers. Updates on T' are performed using the amortized-case updating strategy described in Section 2.2. The correctness of the method is clear and the space and query time bounds follow from Theorem 4.4. Also, from Theorem 4.4, the amortized update time of an auxiliary structure is $\bar{U}(n) = O(\log n)$ and so the amortized update time of the entire structure is $O(\bar{U}(n) \log n) = O(\log^2 n)$.

Lemma 5.1 *Let S be a set of n colored points in the plane. There exists a data structure of size $O(n \log n)$, such that for any grounded query rectangle $[a, b] \times [c, \infty)$, we can report the i distinct colors of the points that are contained in it in $O(\log^2 n + i)$ time. Moreover, if we do n insertions into an initially-empty set then the amortized insertion time is $O(\log^2 n)$.* $\Box$

To solve the problem for $q = [a, b] \times [c, d]$, we use the above approach again, except that we store the points in the tree by sorted y-coordinates and the auxiliary structure at each node of the tree is the one from Lemma 5.1. We conclude directly:

Theorem 5.3 *Let S be a set of n colored points in the plane. There exists a data structure of size $O(n \log^2 n)$ such that for any query rectangle $[a, b] \times [c, d]$, we can report the i distinct colors of the points that are contained in it in $O(\log^2 n + i)$ time. Moreover, if we do n insertions into an initially-empty set then the amortized insertion time is $O(\log^3 n)$.* □

6 Generalized 3-dimensional range searching

The semidynamic structure of Theorem 5.3 coupled with persistence allows us to go up one dimension and solve the following problem: Preprocess a set S of n colored points in 3-space so that for any query box $q = [a, b] \times [c, d] \times [e, f]$ the i distinct colors of the points inside q can be reported efficiently.

First consider queries of the form $q' = [a, b] \times [c, d] \times [e, \infty)$. We sort the points of S by nonincreasing z-coordinates, and insert them in this order into a partially persistent version of the structure of Theorem 5.3, taking only the first two coordinates into account. To answer q', we access the version corresponding to the smallest z-coordinate greater than or equal to e and query it with $[a, b] \times [c, d]$.

Lemma 6.1 *Let S be a set of n colored points in 3-space. S can be stored in a data structure of size $O(n \log^3 n)$ such that for any query box $[a, b] \times [c, d] \times [e, \infty)$, we can report the i distinct colors of the points that are contained in it in $O(\log^2 n + i)$ time.*

Proof First note that the structure of Theorem 5.3 has constant in-degree, so that the results of [DSST89] apply. The correctness of the method and the query time are easy to see. By Theorem 5.3 the total insertion time is $O(n \log^3 n)$, which implies the same bound for the number of memory modifications. Thus the space bound is $O(n \log^3 n)$. □

Now, to solve the problem for $q = [a, b] \times [c, d] \times [e, f]$, we use an approach similar to Lemma 5.1, except that the points are stored in the tree by sorted z-coordinates and the auxiliary structure at each node is the one from Lemma 6.1. However, since the problem is static the tree need not be a $BB(\alpha)$ tree; any static balanced binary tree will do.

Theorem 6.1 *Let S be a set of n colored points in 3-space. S can be stored in a data structure of size $O(n \log^4 n)$ such that for any query box $[a, b] \times [c, d] \times [e, f]$, we can report the i distinct colors of the points that are contained in it in $O(\log^2 n + i)$ time.* □

7 Generalized interval intersection searching

In this section we give some applications of the results of Section 4. Let S be a set of n colored intervals on the x-axis. We show how to preprocess S so that for any query interval $q = [a, b]$, we can solve the static counting, the static reporting, and the dynamic and semi-dynamic reporting problems efficiently.

Our approach is as follows: Let $[x, y]$ be any interval in S. Then $[a, b]$ intersects $[x, y]$ if and only if $x \leq b$ and $y \geq a$. Let us map $[x, y]$ to the point (x, y) in the plane and $[a, b]$ to the query point (b, a). Then it is clear that $[a, b]$ intersects $[x, y]$ if and only if (x, y) lies in the northwest quadrant of (b, a). It follows that we can solve the generalized problems stated above by using a data structure for the corresponding generalized quadrant searching problem (these structures are symmetric to the ones in Theorems 4.1–4.4).

Theorem 7.1 *A set S of n colored intervals on the real line can be preprocessed so that given a query interval q:* **(1)** *The number of distinctly-colored intervals intersected by q can be counted in $O(\log n)$ time using $O(n \log n)$ space.* **(2)** *The i distinct colors of the intervals intersected by q can be reported in $O(\log n + i)$ time using $O(n)$ space.* **(3)** *The i distinct colors of the intervals intersected by q can be reported in $O(\log^2 n + i \log n)$ time and, moreover, intervals can be inserted or deleted online in $O(\log^2 n)$ time. The space used is $O(n \log n)$. In the insertions-only case, the bounds can be improved to $O(n)$ space, $O(\log^2 n + i)$ query time, and $O(\log n)$ insertion time, where the insertion time is amortized over a sequence of n insertions into an initially-empty set.* □

We note that the static reporting version has been solved previously, with the same bounds, in [JL93]. However, our solution above is simpler and more direct. No results were known for the other two problems, although a very restricted version of the dynamic reporting problem has been considered in [Lop91]. Under the assumption that no two intervals of the same color ever overlap, a solution is given in [Lop91] which uses $O(n \log n)$ (resp., $O(n)$) space, $O(\log n + i)$ (resp., $O(\log^2 n + i)$) query time, and $O(\log^2 n)$ (resp., $O(\log n)$) amortized update time.

8 Generalized 1-dimensional point enclosure searching

We show how to preprocess a set, S, of n colored intervals on the x-axis so that for any query point q on the x-axis we can solve efficiently the counting problem (both types) and the dynamic reporting problem. The static reporting problem was solved in $O(n)$ space and $O(\log n + i)$ query time [JL93]. No results were known for the other problems. (The dynamic reporting version is solvable, under the restriction that no two intervals of the same color overlap, using the results in [Lop91] mentioned in Section 7.)

8.1 The counting problem

In preprocessing, we replace all equal-valued left (resp., right) endpoints in S by a single left (resp., right) endpoint of that value. Then we sort the resulting set of endpoints in nondecreasing order into an array A, such that if there is a left endpoint and a right endpoint of the same value, then the left endpoint is stored before the right endpoint. The endpoints in A partition the x-axis into at most $2n+1$ *basic intervals*. For each endpoint, we store the number of distinctly-colored intervals that span the basic interval to its right. We also store with the endpoint a bit indicating whether it is the left endpoint or the right endpoint of an interval in S.

Given q, we locate it in A via binary search. If q falls outside A, we return zero. Otherwise, if q falls between two entries of A or if it coincides with an entry that is the right endpoint of some interval in S, then we return the count associated with the entry to the left of q. Otherwise, q coincides with an entry that is a left endpoint of some interval in S and we return the count associated with that entry.

Theorem 8.1 *A set S of n colored intervals on the real line can be preprocessed into a data structure of size $O(n)$ such that the number of distinctly-colored intervals that contain a query point can be determined in $O(\log n)$ time.*

Proof The correctness of the method is clear if q falls inside a basic interval. Consider the case where q coincides with an endpoint e. Then we need to count the distinct colors of the intervals that (i) contain e, (ii) that begin at e, and (iii) that end at e. Suppose that e is a left endpoint. Then intervals of the first two types span the basic interval to e's right. If there are intervals of the third type also, then they must span the (empty) basic interval to e's right; this follows from the way we broke ties. Thus the query algorithm is correct in returning the count associated with e. Suppose that e is a right endpoint. Intervals of the first and third type span the basic interval to e's left. If intervals of the second type exist, then they too must span the (empty) basic interval to e's left, because of how we broke ties. Thus the query algorithm is correct in returning the count associated with e's predecessor.

The space and query time bounds are clear. □

8.2 The type-2 counting problem

The structure is similar to the one in Section 8.1. However, instead of storing a count with each entry of A, we store a doubly-linked list. The list contains, for each color c, the item $< c, m >$ if there are $m > 0$ intervals of color c spanning the basic interval to the right of the entry. The $O(n^2)$ space implied by this approach can be reduced to $O(n)$ using persistence, as follows: As we march left to right on the x-axis, each list changes from the previous one through (i) insertion of color c if we see the left endpoint of an interval of color c and c is not in the previous list, or (ii) incrementing of c's count if we see the left endpoint of an interval of color c and c is already in the

previous list, or (iii) decrementing of c's count if we see the right endpoint of an interval of color c; this is followed by the deletion of the entry for c if its count becomes zero.

Thus, to create our space-efficient structure, we start with an initially-empty doubly-linked list, scan the endpoints in left to right order, and perform the appropriate operation from above in a partially persistent fashion on the list. Each operation causes only $O(1)$ memory modifications in the current list and so the total space is just $O(n)$. To answer a query, we simply access the appropriate list, using the approach of Section 8.1, and output it.

Theorem 8.2 *A set S of n colored intervals on the real line can be preprocessed into a data structure of size $O(n)$ such that for any query point, a type-2 counting query can be answered in $O(\log n + i)$ time, where i is the output size.* □

8.3 The dynamic reporting problem

The data structure that we use is a *dynamic interval tree*, T, which is implemented as a $BB(\alpha)$ tree [Meh84, pages 192–198]. The distinct x-coordinates of the endpoints of the intervals in S are stored in increasing order from left to right at the leaves of T. For any node v, let $X(v)$ be the average of the x-coordinate stored in the rightmost leaf in v's left subtree and the x-coordinate stored in the leftmost leaf in v's right subtree; for a leaf v, we take $X(v)$ to be the x-coordinate stored at v. Each interval, I, of S is allocated to the highest node v such that I intersects $X(v)$. We call the set of intervals allocated to v the *node-list of* v and denote it by $NL(v)$. The following is the auxiliary structure stored at v:

1. Balanced search trees $NL_l(v)$ and $NL_r(v)$: $NL_l(v)$ stores the left endpoints of the intervals in $NL(v)$ in nondecreasing x-order at the leaves and the leaves are threaded into a linked list. $NL_r(v)$ is defined symmetrically. These lists constitute a sorted representation of $NL(v)$.

2. Balanced search trees $L(v)$ and $R(v)$: For each color c, $L(v)$ contains the leftmost c-colored endpoint from $NL_l(v)$. The entries in $L(v)$ are stored in nondecreasing x-order at the leaves and the leaves are threaded into a linked list. $R(v)$ is defined symmetrically.

3. A balanced search tree $CT(v)$ which stores the distinct colors of the intervals in $NL(v)$. With each color c, we store pointers to the entries of color c in $L(v)$ and $R(v)$. We also store with c a pointer to a balanced search tree $B_c(v)$, which stores in sorted order the endpoints of all the intervals of color c in $NL(v)$.

Given a query point q, we search down T from the root. Let v be the current node. If $q \leq X(v)$ then we scan $L(v)$'s leaves from left to right, list out the colors of the entries that are less than or equal to q, and then search $T(\mathit{left}(v))$ recursively. Otherwise, we proceed symmetrically with $R(v)$ and then search $T(\mathit{right}(v))$ recursively.

We now discuss updates. Consider the insertion of a c-colored interval I. We insert I's endpoints in T and then determine the node v to whose node-list I should belong. We insert I's left endpoint into $NL_l(v)$ and into $B_c(v)$. If I's left endpoint becomes the leftmost endpoint in $B_c(v)$ then we update $L(v)$. Symmetrically, with I's right endpoint. We then rebalance T via rotations. Consider Figure 2 which shows a right rotation about an edge $(u, v) \in T$, and representative intervals α, β, and γ. (The discussion for a left rotation and a double rotation is similar.) Before the rotation, $\alpha \in NL(v)$ and $\beta, \gamma \in NL(u)$, whereas after the rotation, $\alpha, \gamma \in NL(v)$ and $\beta \in NL(u)$; as shown in [Meh84], no other node-list is affected by this rotation. Thus, following the rotation, we rebuild, all at once, the auxiliary structures of u and v. A similar discussion applies for the deletion of an interval. (Note that since the rebuilding of auxiliary structures is done all at once, T is always up-to-date for a query and so the query algorithm above applies unchanged.)

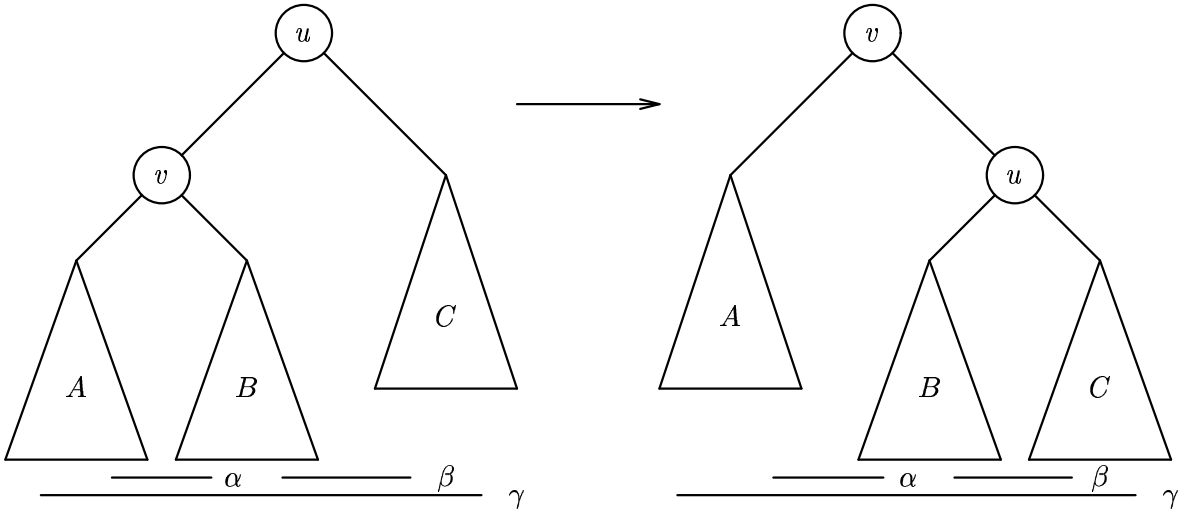

Figure 2: *Single right rotation and representative intervals α, β, and γ.*

Theorem 8.3 *Let S be a set of n colored intervals on the real line. S can be preprocessed into a data structure of size $O(n)$ such that the i distinct colors of the intervals of S that contain a query point can be reported in $O((i+1)\log n)$ time and intervals can be inserted and deleted online in $O(\log n)$ time, where the update time is amortized over a sequence of n updates into an initially-empty set.*

Proof We show that the query algorithm is correct. Note first that if a node is not on the search path then no interval in its node-list can contain q. This is because for any node in T, the endpoints of the intervals in its node-list are contained in its subtree and the search guarantees that if a node

is not on the search path then q is not in the range of x-coordinates stored in its subtree. Thus only nodes v on the search path need be considered.

We claim that a color c is reported at a node v if and only if a c-colored interval $I \in NL(v)$ contains q. W.l.o.g. assume $q \leq X(v)$. If I contains q, then q is greater than or equal to the left endpoint of I and hence greater than or equal to the left endpoint of I', where $I' \in NL(v)$ is the c-colored interval whose left endpoint appears in $L(v)$. Thus I''s left endpoint will be encountered in the scan of $L(v)$ and color c will be reported. For the converse, assume that color c is reported. Then the left endpoint of I' must have been encountered in the scan of $L(v)$ and so q is greater than or equal to this endpoint. Moreover, since $I' \in NL(v)$, its right endpoint must be in v's right subtree and hence must be greater than or equal to q. Thus I' contains q.

The query time at any node is $O(i) + O(1)$ and so the overall query time is $O((i+1)\log n)$. The space used by the structure is $O(n)$ since each interval is in exactly one node-list and the space used by the auxiliary structure of a node is linear in the size of the corresponding node-list.

Consider the insertion or deletion of an interval I. The time to update the auxiliary structure of the node to whose node-list I belongs is clearly $O(\log n)$. Next, consider the time for doing the rebuilding at the nodes u and v that are involved in the rotation. Given the sorted representations of the old $NL_l(u)$ and $NL_l(v)$, we can construct the new $NL_l(u)$ and $NL_l(v)$ in sorted order in $O(|NL(u)| + |NL(v)|)$ time, as follows: We merge the lists of leaves of the old $NL_l(u)$ and $NL_l(v)$ into a single list, then extract from this, in sorted order, the lists of leaves for the new $NL_l(u)$ and $NL_l(v)$ (by checking whether the interval corresponding to a leaf belongs to the new $NL(u)$ or to the new $NL(v)$), and then build the trees on these two lists bottom-up. Similarly for $NL_r(u)$ and $NL_r(v)$. Once this is done, L, R, and CT at u and v can be easily constructed in the same amount of time. Thus the cost of a rotation is $O(|NL(u)| + |NL(v)|)$. It then follows from the results in [Meh84] that the update time is $O(\log n)$, when amortized over a sequence of n updates into an initially-empty set S. $\square$

9 Generalized 2-dimensional point enclosure searching

We show how to preprocess a set S of n colored, axes-parallel rectangles in the plane so that given a query point $q = (a, b)$, we can solve efficiently the static reporting and counting problems. For the reporting problem, an $O(n^{1.5})$-space and $O(\log n + i)$-query time structure was given in [JL93]. No results were known for the counting problem.

9.1 The reporting problem

We create a *segment tree*, T, [Meh84, pages 212–215], on the distinct x-coordinates of the vertical sides of the rectangles in S. Each node of T will contain a data structure solving the generalized one-dimensional point enclosure problem for an appropriate set of y-intervals, which we now define.

Let $r = [x_1, x_2] \times [y_1, y_2]$ be a rectangle of S. Let v be a node of T such that the range of v is contained in $[x_1, x_2]$, but the range of v's parent is not. Then, rectangle r—or more precisely, the interval $[y_1, y_2]$—is *associated* with v. (However, we do not necessarily store $[y_1, y_2]$ at v.)

Now we define the set of y-intervals that are *allocated* to v—these are the intervals on which the auxiliary structure of v is built. If v is the root of T then the intervals allocated to it are just the intervals associated with it. If v is a non-root node of T, then let $v_1, v_2, \ldots, v_m = v$ be the nodes on the path from the root, v_1, to v. For each color c, let $s_1, \ldots, s_k$ be the pairwise disjoint y-intervals that form the union of all c-colored y-intervals that are allocated to $v_1, \ldots, v_{m-1}$. Moreover, let $t_1, \ldots, t_l$ be the pairwise disjoint y-intervals that form the union of all c-colored y-intervals that are associated with v. Then, we allocate to v the pairwise disjoint c-colored y-intervals that span

$$\left(\bigcup_{i=1}^{l} t_i\right) \setminus \left(\bigcup_{i=1}^{k} s_i\right).$$

We take the intervals of all colors allocated to v and store them in the data structure for generalized 1-dimensional point enclosure reporting given in [JL93]. If n_v is the number of intervals allocated to v, then this structure uses $O(n_v)$ space and reports the i_v distinct colors of the intervals containing a query point in $O(\log n_v + i_v)$ time.

Given a query point $q = (a, b)$, we proceed as follows: Assume that a is in the range of the root of T; otherwise, we can stop. We do a binary search in T for a and query with b the auxiliary structure of each node v visited.

Theorem 9.1 *Let S be a set of n colored, axes-parallel rectangles in the plane. S can be stored in a data structure of size $O(n \log n)$ so that for any query point q, the i distinct colors of the rectangles containing q can be reported in $O(\log^2 n + i)$ time.*

Proof We first prove the correctness of the query algorithm. The search visits a node of T if and only if the node's range contains a. Thus, the rectangles associated with the nodes on the search path are precisely the ones whose x-intervals contain a. Any such rectangle contains q if and only if its y-interval contains b. Thus we can report the distinct colors of the rectangles containing q by doing at each node v on the search path a generalized 1-dimensional point enclosure query on the y-intervals associated with v, using b as the query point. Recall though that our auxiliary structures are built on the y-intervals allocated to v. However, the portion of any y-interval associated with v that contains b is allocated to v or to a predecessor of v on the search path. It follows that the distinct colors of the rectangles containing q will be reported.

We now establish the query time. Note that, for any c, the c-colored intervals allocated to v are pairwise disjoint. Moreover, if v and w are nodes of T such that v is in w's subtree, then the c-colored intervals allocated to v and to w are pairwise disjoint, except possibly at endpoints. Thus there are at most two c-colored allocated intervals on the search path that contain b and so the

total number of distinctly-colored intervals intersected during the query is at most $2i$. In addition, $O(\log n)$ time is spent in searching the auxiliary structure at each of $O(\log n)$ nodes visited in the search. Thus the query time is $O(\log^2 n + i)$. (Note that since the auxiliary structure used in [JL93] is a window list [Cha86], it is not clear how to reduce the query time by applying fractional cascading.)

Finally, we prove the space bound. Since each rectangle is associated with $O(\log n)$ nodes of T [Meh84], the total number of y-intervals associated with nodes of T is $O(n \log n)$. We will prove that the total number of allocated intervals is also $O(n \log n)$ by counting the total number of endpoints of allocated intervals.

Call an endpoint of an allocated interval a *primary* endpoint if it is an endpoint of an interval associated with some node of T. Call it a *secondary* endpoint if it is an endpoint created when a primary endpoint in $\cup s_i$ chops off a portion of some interval in $\cup t_i$. Note that secondary endpoints can be created only by primary endpoints; a secondary endpoint created at a node v cannot create a secondary endpoint at any descendant u of v since at u it will be in the interior of some interval in $\cup s_i$.

The total number of primary endpoints is clearly $O(n \log n)$. How many secondary endpoints are there? Observe that: (1) If e is the endpoint of an interval associated with v then e could be a primary endpoint of an interval allocated to v but e cannot be an endpoint of any interval allocated to a descendant u of v since intervals allocated to v are subtracted out at u. (2) e can create at most one secondary endpoint in v's left subtree; if it creates a secondary endpoint at a node w in v's left subtree, then at any descendant w' of w, e will fall in the interior of some interval in $\cup s_i$. Similarly for v's right subtree.

We charge each secondary endpoint to the primary endpoint that created it. By the above discussion, each primary endpoint is charged $O(1)$. Thus there are $O(n \log n)$ secondary endpoints in total and hence $O(n \log n)$ endpoints of allocated intervals. It follows that there are $O(n \log n)$ allocated intervals.

As mentioned before, if there are n_v intervals allocated to v, then the auxiliary structure at v uses $O(n_v)$ space. Since an allocated interval appears in exactly one node and there are $O(n \log n)$ of them, it follows that the auxiliary structures occupy a total of $O(n \log n)$ space. □

9.2 The counting problem

Except for two key differences, the approach described in Section 9.1 carries over to the counting problem once we replace the reporting version of the auxiliary structure by the counting version (Theorem 8.1) and add up the counts reported from each auxiliary structure queried.

The first difference is that in the reporting version we could afford to allow a c-colored interval allocated to a node w to share an endpoint with a c-colored interval allocated to a descendant v. However, we cannot allow this in the counting version, since then c would be counted twice if q's

y-coordinate coincided with that endpoint. To remedy this, we exclude the shared endpoint in the c-colored interval allocated to the descendant v. Thus the auxiliary structure of a node can now contain closed intervals, half-open intervals, and open intervals. The structure of Theorem 8.1 can be modified very easily to handle this without affecting its performance.

The second difference is that we can now apply fractional cascading to the auxiliary structures so that the query time can be reduced to $O(\log n)$. (Actually, we do not apply fractional cascading directly to the auxiliary structures since the cascaded elements can change the answer. Instead, we use the approach described in Section 3.1.2: at each node we create a list, which is used to guide the search in the auxiliary structure, and then apply fractional cascading to these lists.)

Theorem 9.2 *A set S of n colored, axes-parallel rectangles in the plane can be preprocessed into a data structure of size $O(n \log n)$ such that the number of distinctly-colored rectangles that contain a query point can be determined in $O(\log n)$ time.* □

10 Generalized orthogonal segment intersection searching

We show how to preprocess a set, S, of n colored horizontal line segments in the plane such that the number of distinctly-colored segments intersected by a vertical query segment q can be determined efficiently.

We sort the endpoints of S by non-decreasing x-coordinate. A tie between two left or two right endpoints is broken arbitrarily; a tie between a left endpoint and a right endpoint is broken by storing the left endpoint before the right endpoint. Then we sweep over the sorted list. When we encounter a left endpoint, we insert its y-coordinate (which is given the same color as the endpoint) into a partially persistent version of the structure of Theorem 3.3 for dynamic generalized 1-dimensional range counting. Similarly, when we see a right endpoint, we delete it's y-coordinate from the structure.

We answer a query as follows: Let a be q's x-coordinate and let $e_1, e_2, \ldots, e_{2n}$ be the endpoints in sorted order. If a falls between e_{i-1} and e_i, then we query the version corresponding to e_{i-1}. Suppose that a coincides with an e_i. If e_i is a left endpoint, then we determine the rightmost left endpoint e_j with the same x-coordinate as e_i (this can be done in $O(1)$ time by storing, during the preprocessing, a pointer with e_i) and query the version corresponding to e_j. If e_i is a right endpoint, then we determine the leftmost right endpoint e_j with the same x-coordinate as e_i and query the version for e_{j-1}.

Using an argument similar to the one in the proof of Theorem 8.1, it can be shown that the version that is accessed contains exactly those segments of S that include q's x-coordinate. Thus the problem becomes 1-dimensional in y. Since the structure of Theorem 3.3 supports updates in $O(\log^2 n)$ time, the number of memory modifications per update is $O(\log^2 n)$. Thus the persistent structure occupies $O(n \log^2 n)$ space and supports queries in $O(\log^2 n)$ time.

Theorem 10.1 *A set S of n colored horizontal line segments in the plane can be preprocessed into a data structure of size $O(n \log^2 n)$ such that the number of distinctly-colored segments that are intersected by a vertical query segment can be determined in time $O(\log^2 n)$.* □

11 Conclusions and open problems

We have considered a class of problems that generalizes the class of standard intersection searching problems and is rich in applications. We have presented a uniform framework to solve efficiently a wide variety of problems in this class. Our techniques have been based on persistent data structures and, to a somewhat lesser extent, on a geometric transformation.

Several open problems remain. First, can other problems, such as, for instance, generalized segment intersection searching, be solved using the techniques given in this paper? Second, there are several gaps in Table 2 that need to be filled, especially with regard to dynamic counting and reporting problems. Finally, it would be desirable to improve the query time of $O(\log^2 n + i \log n)$ for some of the fully dynamic solutions in Table 2 to $O(\text{polylog}(n) + i)$.

Acknowledgement

The counting problem was initially posed to one of the authors (RJ) by Matthew Katz at a summer school in Computational Geometry, which was held at the Fibonacci Institute in Trento, Italy, from June 15–19, 1992.

The authors would like to thank the two referees for numerous suggestions that helped improve the paper.

References

[CG86] B.M. Chazelle and L.J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[Cha86] B.M. Chazelle. Filtering search: a new approach to query-answering. *SIAM Journal on Computing*, 15:703–724, 1986.

[CJ90] S.W. Cheng and R. Janardan. Efficient dynamic algorithms for some geometric intersection problems. *Information Processing Letters*, 36:251–258, 1990.

[CJ92] S.W. Cheng and R. Janardan. Algorithms for ray-shooting and intersection searching. *Journal of Algorithms*, 13:670–692, 1992.

[DSST89] J.R. Driscoll, N. Sarnak, D.D. Sleator, and R.E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[EKM82] H. Edelsbrunner, D. Kirkpatrick, and H.A. Maurer. Polygonal intersection searching. *Information Processing Letters*, 14:74–79, 1982.

[EM81] H. Edelsbrunner and H. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 13:177–181, 1981.

[JL93] R. Janardan and M. Lopez. Generalized intersection searching problems. *International Journal on Computational Geometry & Applications*, 3(1):39–69, 1993.

[Lop91] M. Lopez. *Algorithms for composite geometric objects.* PhD thesis, Department of Computer Science, University of Minnesota, Minneapolis, Minnesota, 1991.

[LP84] D.T. Lee and F.P. Preparata. Computational geometry – a survey. *IEEE Transactions on Computers*, 33:1072–1101, 1984.

[McC85] E.M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14:257–276, 1985.

[Meh84] K. Mehlhorn. *Data structures and algorithms 3: Multi-dimensional searching and computational geometry.* Springer–Verlag, 1984.

[NR73] J. Nievergelt and E.M. Reingold. Binary search trees of bounded balance. *SIAM Journal on Computing*, 2:33–43, 1973.

[Ove83] M.H. Overmars. *The design of dynamic data structures.* Springer–Verlag, 1983.

[PS88] F.P. Preparata and M.I. Shamos. *Computational geometry – an introduction.* Springer–Verlag, 1988.

[SB79] J.B. Saxe and J.L. Bentley. Transforming static data structures into dynamic data structures. In *Proceedings of the 20th Annual IEEE Symposium on Foundations of Computer Science*, pages 148–168, 1979.

[VW82] V.K. Vaishnavi and D. Wood. Rectilinear segment intersection, layered segment trees, and dynamization. *Journal of Algorithms*, 3:160–176, 1982.

[WL85] D.E. Willard and G.S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32:597–617, 1985.