

Optimal and Near-Optimal Algorithms for Generalized Intersection Reporting on Pointer Machines*

(Technical Report CS-TR-4542 and UMIACS-TR-2003-110)

Qingmin Shi and Joseph JaJa
*Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742*
{*qshi,joseph@umiacs.umd.edu*}

November 18, 2003

Abstract

We develop efficient algorithms for a number of generalized intersection reporting problems, including orthogonal and general segment intersection, 2-D range searching, rectangular point enclosure, and rectangle intersection search. Our results for orthogonal and general segment intersection, 3-sided 2-D range searching, and rectangular pointer enclosure problems match the lower bounds for their corresponding standard versions under the pointer machine model. Our results for the remaining problems improve upon the best known previous algorithms.

1 Introduction

Let S be a set of n colored geometric objects in \mathcal{R}^d of the same type, e.g., points, segments, rectangles. A *generalized intersection searching* problem is defined as the task of preprocessing S so that, given a query object q , the distinct colors of the objects in S that intersect q can be reported quickly. Problems of this type occur frequently in various areas such as VLSI design, database retrieval, text analysis, and network traffic analysis (see [JL93, Mut02, AGM02] for a description of a number of related applications).

The standard intersection reporting problems, studied extensively over the past two decades, are special cases of their corresponding generalized versions, since the objects can be viewed as having distinct colors.

Janardan and Lopez [JL93] were the first to consider the generalized intersection searching problems. They described algorithms to handle fundamental problems, e.g. segment

*Supported in part by the National Science Foundation through the National Partnership for Advanced Computational Infrastructure (NPACI), DoD-MD Procurement under contract MDA90402C0428, and NASA under the ESIP Program NCC5300.

intersection, range searching, and rectangular point enclosure. In [GJS95], Gupta, Janardan, and Smid studied the dynamic case in which objects can be inserted into and deleted from S , as well as considered the counting versions of these problems. Results for generalized intersection problems involving non-iso-oriented objects, i.e., objects whose boundaries are not parallel to the axes, can be found in [AvKO93] and [GJS94]. Special versions of the generalized range search, where the boundaries of the objects are supported by an integer grid $\{1, \dots, U\}^d$, have been studied in [Mut02, AGM02, Mor03] under the RAM model. A recent survey of generalized intersect searching algorithms can be found in [GJS].

The performance measure of interest for generalized intersection searching problem are the space used by the data structure generated during the preprocessing step and the query time. An interesting open problem raised by Janardan and Lopez's paper [JL93] is whether it is possible to solve the generalized intersection problems within the same space and time bounds as their standard counterparts. In this paper, we show that this is indeed the case for almost all the problems studied in [JL93].

The main results of this paper and their comparisons to the best previous results are given in Tables 1, 2, 3, and 4 in the case when the coordinates are from \mathcal{R}^2 . For completeness, we also include previous results¹ that are only valid under the stronger RAM model.

Model	Query time	Space	Source
Pointer	$O(\log n + k)$	$O(n \log n)$	[JL93]
Pointer	$O(\log^2 n + k)$	$O(n)$	[JL93]
RAM	$O(\log n + k)$	$O(n \log \log n)$	[Mor03]
RAM	$O(\log n \log^2 \log n + k)$	$O(n)$	[Mor03]
Pointer	$O(\log n + k)$	$O(n)$	New

Table 1: Generalized orthogonal segment intersection

Model	Query time	Space	Source
Pointer	$O((k+1)\sqrt{n} \log^{O(1)} n)$	$O(n^{1+\epsilon})$	[AvKO93]
Pointer	$O(\log^2 n + k \log n)$	$O(n \log n)$	[GJS94]
Pointer	$O(\log n + k)$	$O(n^2 \log n)$	[JL93]
Pointer	$O(\log^2 n + k)$	$O(n^2)$	[JL93]
Pointer	$O(\log n + k)$	$O(n^2)$	New

Table 2: Generalized segment intersection (non-intersecting line segments)

Our results for the generalized orthogonal and general segment intersection and rectangular point enclosure problems match the lower bounds under the pointer machine model. The lower bound for the general segment intersection problem was shown by Chazelle

¹One of Mortensen's results in [Mor03] for generalized orthogonal segment intersection on an $\{1, \dots, n\}^2$ grid is $O(\log^2 \log n + k)$ query time using $O(n \log \log n)$ space. One of his results [Mor03] for generalized 2-D range reporting is $O(\log^2 \log n + k)$ time using $O(n \log n \log \log n)$ space. Agarwal, Gobindarajan, and Muthukrishnan's result [AGM02] for generalized rectangular point enclosure problem on an $\{1, \dots, U\}^2$ grid is $O(\log \log U + k)$ query time using $O(n^{1+\epsilon})$ space, where ϵ is an arbitrarily small positive number. These results as cited in Tables 1, 3, and 4 have been translated to handle problems in \mathcal{R}^2 .

Model	Query time	Space	Source
Pointer	$O(\log^2 n + k)$	$O(n \log n)$	[JL93]
Pointer	$O(\log n + k)$	$O(n \log^2 n)$	[JL93]
RAM	$O(\log n + k)$	$O(n \log n \log \log n)$	[Mor03]
RAM	$O(\log n \log^2 \log n + k)$	$O(n \log n)$	[Mor03]
Pointer	$O(\log n + k)$	$O(n \log n)$	New

Table 3: Generalized 2-D range reporting

Model	Query time	Space	Source
RAM	$O(\log n + k)$	$O(n^{1+\epsilon})$	[AGM02]
Pointer	$O(\log^2 n + k)$	$O(n \log n)$	[GJS95]
Pointer	$O(\log n + k)$	$O(n)$	new

Table 4: Generalized rectangular point enclosure

and Liu in [CL01]. For the standard 2-D range searching problem, the lower bound is $\Omega(n \log n / \log \log n)$ space and $\Omega(\log n + k)$ query time [Cha90], which is the only lower bound we could not match.

The algorithms discussed in this paper use the so-called *linked data structure* introduced by Driscoll et al. in [DSST89]. Throughout this paper, we will largely adopt the terminology used in that paper. A linked data structure consists of a set of interconnected nodes with a fixed number of fields. Each field holds either a single piece of information or a pointer to another node. Access to a linked data structure starts with an *entry node*, whose access is provided by an *access pointer*. A query on a linked data structure is called an *access operation*, which consists of a sequence of *access steps*. A node can be accessed only if it is either an entry node or a node pointed to by a pointer field of a node that has already been accessed. Insertion and deletion operations on a linked data structure are called *update operations*, which consist of an intermixed sequence of access steps and *update steps*. The access steps locate the parts of the data structure that need to be changed and the update steps actually change the structure either by creating a new node or by modifying the fields of a node that has already been accessed or modifying an access pointer. Note that access and update operations as described above are allowed on a pointer machine.

Persistent linked data structures differ from ordinary (*ephemeral*) linked data structures in that after a sequence of update operation, the old versions can still be accessed. Our algorithms are based on the persistence technique of Driscoll et al. [DSST89], which states that any ephemeral linked data structure with constant in-degree can be made persistent such that each update step of the ephemeral data structure adds $O(1)$ amortized space to the persistent data structure and a query on any version of the persistent data structure can be handled within the same time bound as on the ephemeral structure, provided that the access pointer corresponding to that version can be computed in constant time. It is easy to verify that the ephemeral versions of the data structures presented in this paper are all linked data structures with constant in-degree and therefore the results of [DSST89] can be applied. Also, the constraint of constant access time to an access pointer can be removed by using a balanced binary tree to index the access pointers by their corresponding versions

and, as a result, adding an $O(n)$ term to the space and an $O(\log n)$ term to the query time, where n is number of objects stored in the persistent data structure.

Our approach to handle a generalized intersection searching problem consists of designing a dynamic data structure for a simpler problem (with less constraints), which supports efficient query as well as update operations, followed by making this data structure partially persistent to solve the original problem. This idea has been exploited previously (for example in [GJS94, GJS95, BM95, AGM02, Mor03]). However, what leads to our better results is the key observation that it is not necessary for the solution to the simpler problem to be the best in terms of space and query time. What is important is for this solution to support update operations with a small number of update steps. With this observation in mind, we start from the final persistent data structure rather than making an existing ephemeral data structure persistent. Since the size of the final data structure depends on the total number n of objects stored in it, we tolerate the space cost and the query complexity that are functions of n instead the number of objects stored in the data structure. In exchange, we are able to obtain a more “static” ephemeral data structure, which requires less update steps in an update operation.

In this paper, we will always use k to denote the size of the set of reported colors. We use $(x_1, x_2; y)$, with $x_1 < x_2$, to represent a horizontal segment, whose two endpoints are (x_1, y) and (x_2, y) . A vertical segment is denoted as $(x; y_1, y_2)$, with $y_1 < y_2$, (x, y_1) and (x, y_2) being its two endpoints. Similarly, we use $(x_1, x_2; y_1, y_2)$ to represent an iso-oriented rectangle whose projections to the x- and y-axes are $[x_1, x_2]$ and $[y_1, y_2]$, respectively.

2 Generalized 1-D Range Search

We start by investigating the generalized 1-D range searching problem. Our solution will serve as a foundation for a number of 2-D generalized intersection problems to be discussed later.

Optimal algorithms for the generalized 1-D range searching problems have been developed by Jarnardan and Lopez [JL93] and Gupta, Jarnardan, and Smid [GJS94]. For m colored one-dimensional points, the k distinct colors of those points that fall within an interval can be reported in $O(\log m + k)$ time using $O(m)$ space. The latter algorithm can be made dynamic to support each insertion or deletion operation with $O(\log m)$ update steps.

Since our ultimate goal is to make the data structure partially persistent to handle more complicated problems, we aim at minimizing the number of update steps rather than space and query time. More specifically, we develop a dynamic data structure for the generalized 1-D range searching problem, which supports insertion and deletion operations with only $O(1)$ update steps but with $O(n)$ space. This will enable us to derive $O(n)$ space persistent versions to solve the 2-D generalized intersection searching problem.

We define the following *dynamic generalized 1-D range searching* problem.

Problem 2.1. *Let U be a set of n real numbers and let S be a set of $m \leq n$ colored 1-D points whose coordinates are taken from U . Preprocess S so that (i) given a query specified by (a, b) , $a < b$, the k distinct colors of the points $p = (p.x)$ in S that satisfy $a \leq p.x \leq b$ can be reported in $O(\log n + k)$ time; (ii) a new point whose coordinate is also from U can be*

inserted into the existing data structure with $O(1)$ update steps; and (iii) an existing point in S can be removed with $O(1)$ update steps.

In the above problem, the set U of real numbers consists of the coordinates of the objects in the corresponding persistent data structure, and S is the set of the coordinates of the objects in each ephemeral version of the data structure.

In [Mor03], Mortensen gave a reduction of a generalized 1-D range reporting problem on an integer grid to a special case of the standard 3-sided 2-D range reporting problem. The reduction described in Section 2.1 is largely his with the only difference that we build a balanced tree on U instead of the set S .

2.1 Reduction to 3-Sided 2-D Reporting

We build a balanced binary tree T on the real numbers in U sorted in increasing order. For each node v in T , let S_v be the subset of S stored in the subtree T_v rooted at v , and let $left(v)$ and $right(v)$ respectively denote the leftmost and rightmost leaf nodes in T_v . We will use $height(v)$ to denote the height of v (the height of a leaf node is 0). Given a query specified by (a, b) , we can in $O(\log n)$ time find the two leaf nodes u and v of T that correspond to the successor of a and the predecessor of b in U respectively, and then in $O(\log n)$ time find the nearest common ancestor w of u and v . Let r be the left child of w and t be the right child of w . To answer the query (a, b) , we only need to answer the two generalized 1-D range queries $(a, right(r))$ and $(left(t), b)$. Note that each color will be reported at most twice as long as each color is reported at most once for each of these two queries.

Since the above two queries are symmetric, we only discuss the data structure used to answer the queries of the type $(left(r), b)$ and the related complexity of inserting new points and removing existing points. For an internal node v , we define L_v to be a subset of S_v so that a point p belongs to $L(v)$ if and only if, for every point $p' \neq p$ in S_v that satisfies $p'.x < p.x$, its color is different from that of p . In other words, p is the leftmost point in S_v among those with the same color as p . It is obvious that the number of points in L_v is equal to the number of distinct colors associated with the points in S_v .

Let u be the leaf node corresponding to a point p . We assign a y-coordinate $p.y$ to point p as follows. Let v be the highest ancestor of u such that $p \in L_v$. Then $p.y = height(v)$. Note that $p.y$ is an integer from $\{0, 1, \dots, H\}$, where $H = \lceil \log_2 n \rceil$. By doing so, we convert the set S of colored 1-D points to a set S' of 2-D points without colors.

A generalized 1-D range query $(left(r), b)$ on S can be handled as follows. Let h be the height of v . Then the answer to the query is given by the following standard 3-sided 2-D range queries on S' : report the points p in S' that satisfy $a \leq p.x \leq b$ and $p.y \geq h$.

Lemma 2.1. *The above algorithm correctly determines the distinct colors of the points falling within the query range. Furthermore, each such color is reported once.*

Proof. Suppose there is at least one point in S whose coordinate is within the range $[left(r), b]$ and whose color is c . Let p be the leftmost such point. Obviously, any point in T_r with color c is to the right of p . Therefore, $p.y \geq h$, which means c will be reported. For any other point p' with color c , which is also in the range $[left(r), b]$, $p'.x > p.x$, and hence we have $p'.y < p.y$. Therefore p' will not be reported. \square

We now show that a deletion or insertion involves only a constant number of modifications of the points in S' . We first show how to insert a point p of color c into S . Note that $p.x \in U$. Let v be the leaf node corresponding to $p.x$ and let u be the nearest ancestor of v such that there is at least one point in S_u other than p that has color c . If u does not exist, then $p.y = \lceil \log_2 n \rceil$. Otherwise, let p' be the leftmost such point. If $p'.x < p.x$, which means all the existing points with color c in S_u are to the left of p , then there is no need to update the y-coordinates of any points already in T , and we set $p.y$ to $\text{height}(u) - 1$. If $p'.x > p.x$, we set $p.y$ to $p'.y$ and $p'.y$ to $\text{height}(u) - 1$.

We next describe how to delete a point p of color c from S . Again, let u be the nearest ancestor of the leaf node corresponding to p such that there is at least one point in S_u other than p , which has color c . If u does not exist we simply remove p . Otherwise, let p' be the leftmost such point. If $p'.x < p.x$, we only need to remove p . If $p'.x > p.x$, we set $p'.y$ to $p.y$ and then remove p .

2.2 A Dynamic Generalized 1-D Range Searching Structure Supporting Insertion and Deletion with $O(1)$ Update Steps

The above discussion shows that Problem 2.1 can be reduced to a standard 3-sided 2-D range searching problem by transforming the set S of colored 1-D points into a set S' of 2-D points without color, whose y-coordinates take values from $\{0, \dots, \lceil \log_2 n \rceil\}$. Also notice that an insertion or deletion in Problem 2.1 can be implemented as a constant number of insertions and deletions on S' . Such a 3-sided 2-D range query problem is formally defined as follows.

Problem 2.2. *Let U be a set of n real numbers and let S' be a set of $m \leq n$ 2-D points whose x-coordinates are taken from U and whose y-coordinates are from $\{0, 1, \dots, H\}$, $H = \lceil \log_2 n \rceil$. Preprocess S' so that (i) given a query specified by (a, b, h) , with $a < b$ and $h \in \{0, 1, \dots, H\}$, the k points $p = (p.x, p.y)$ in S' that satisfy $a \leq p.x \leq b$ and $p.y \geq h$ can be reported in $O(\log n + k)$ time; (ii) a new point $p = (p.x, p.y)$, with $p.x \in U$ and $p.y \in \{0, 1, \dots, H\}$ can be inserted into the existing data structure with $O(1)$ update steps; and (iii) a point in S' can be removed from the existing data structure with $O(1)$ update steps.*

Theorem 1. *There exists a data structure such that a query as described in Problem 2.2 can be handled in $O(n)$ space and $O(\log n + k)$ time, where k is the output size. Furthermore, each update operation of this data structure only involves a constant number of update steps.*

The rest of this section is devoted to proving this theorem. A special feature of Problem 2.2 when compared to the most general 3-sided 2-D range query is that the y-coordinates of the points involved can only take one of $H + 1$ values. We partition S' into $H + 1$ subsets L_0, L_1, \dots, L_H so that L_i consists of those points whose y-coordinates are i . The elements in each of these lists are sorted in increasing order of their x-coordinates and chained together as a doubly linked list. We will be able to handle the query described in Problem 2.2 in $O(\log n + k)$ time as long as we spend only $O(1 + k_i)$ time at each subset L_i , where k_i is the number of points in L_i that satisfy the query. We will show in a moment how to achieve the $O(1 + k_i)$ query time for each list by using $O(\log n + k)$ time to find in each list L_i any point p_i that satisfies $a \leq p_i.x \leq b$. Now suppose this is the case. If $p_i.y < h$, we simply skip list

L_i and do not report any point. Otherwise, we scan L_i in two directions, starting from p_i , to report the points in L_i that are in the range $[a, b]$.

We now describe how to find for each of the $H + 1$ lists any point whose x-coordinate is in $[a, b]$. We reduce this problem to a stabbing query on a set W of intervals constructed from the points in S' as follows. For each list L_i , if $L_i = \emptyset$, then it does not contribute any interval to W . Otherwise, let x_1, x_2, \dots, x_j be the list of x-coordinates of the points in L_i sorted in increasing order. The list L_i contributes the following set W_i of $j + 1$ intervals to W : $W_i = \{[x_0, x_1), [x_1, x_2), \dots, [x_j, x_{j+1})\}$ ($W_i = \emptyset$ if $L_i = \emptyset$), where $x_0 = -\infty$ and $x_{j+1} = +\infty$. Obviously, the total number of intervals in W is $m + H + 1$. To find any point in L_i that is within the range of $[a, b]$, we first find the interval in W_i that contains b . Since the intervals in W_i do not overlap, there is exactly one interval in W_i that intersects b , if $W_i \neq \emptyset$. Let it be $[x_i, x_{i+1})$. If $x_i < a$, then no point in L_i should be reported. Otherwise, $x_i \in [a, b]$ and hence we have found a point in L_i . Actually, we do not have to handle a stabbing query for each of the interval sets W_i . Instead, we only need to handle a stabbing query on the entire interval set W .

Such a stabbing query on $m + H + 1$ intervals can be handled using the interval tree technique. By the definition of Problem 2.2, the coordinates of the endpoints of all possible intervals are from U . Therefore, we can keep the skeleton of the interval tree fixed. Adding a point to S' or deleting a point from S' involves (i) inserting that point into or removing that point from one of the doubly linked lists L_i ; and (ii) a constant number of insertions and deletions of intervals to and from the interval tree. Each such operation requires only a constant number of update steps. Thus we have proved Theorem 1 and Corollary 2.1 follows immediately.

Corollary 2.1. *There exists a data structure such that a query described in Problem 2.1 can be handled in $O(n)$ space and $O(\log n + k)$, where k is the output size. Furthermore, each update operation of this data structure only involves a constant number of update steps.*

3 Orthogonal Segment Intersection

The generalized orthogonal segment intersection problem is defined as follows. Preprocess a set P of n colored vertical segments such that, given a horizontal query segment $q = (a, b; c)$, the distinct colors of those vertical segments that intersect q can be reported quickly. Without loss of generality, we assume that the endpoints of all the segments in P have different x- and y-coordinates. We will call the endpoint (x, y_2) of a segment $s = (x; y_1, y_2)$ in S the *head* of s and the endpoint (x, y_1) the *tail* of s .

We solve the generalized orthogonal segment intersection using a partially persistent version of the data structure of Corollary 2.1, which we denote as D , by letting U be the set of x-coordinates of the segments in S . We use an imaginary horizontal line to sweep through these segments from top down. When we encounter the head of a segment s , we insert the x-coordinate of s into D , and assign to it the same color as s . When we encounter the tail of a segment s , we remove the x-coordinate of s from D . Each such update operation is associated with a version ID which is the y-coordinate of the sweeping horizontal line at the time of that operation. Since each such update operation requires only a constant number of update steps, the size of the partially persistent version of D remains $O(n)$.

Given a query $q = (a, b; c)$, the segments in P that intersect q are those whose x -coordinates are stored in the version of D corresponding to the predecessor of c in the sorted list of versions, and fall within the range $[a, b]$. We thus can find the colors of these segments in $O(\log n + k)$ time.

Theorem 2. *There exists a data structure of size $O(n)$ for a set P of n colored vertical segments in \mathcal{R}^2 such that given a horizontal query segment q , the distinct colors of the segments in P that intersect q can be reported in $O(\log n + k)$ time.*

Theorem 2 immediately leads to an optimal algorithm to the following generalized 3-sided 2-D range searching problem: preprocessing a set P of colored 2-D points such that given a query (a, b, c) with $a < b$, the colors of the points $p = (p.x, p.y)$, which satisfy $a \leq p.x \leq b$ and $p.y \geq c$, can be reported quickly. This is due to the fact that each point $p = (p.x, p.y)$ in P can be viewed as a semi-infinite vertical segment $(p.x; -\infty, p.y)$ with the same color as p and the query can be viewed as to find the colors of those vertical segments which intersect the horizontal segment $(a, b; c)$.

Corollary 3.1. *There exists a data structure of size $O(n)$ for a set P of n colored 2-D points in \mathcal{R}^2 such that a generalized 3-sided range query on P can be handled in $O(\log n + k)$ time.*

4 General Segment Intersection

In this section, we discuss the problem of handling the general segment intersection search problem. The set of n colored segments in \mathcal{R}^2 of a set P are not necessarily axes-parallel. The only restriction is that no two segments cross each other (they may touch each other at the endpoints). The direction of a query segment can be arbitrary.

The main framework of our algorithm is the same as that described in [Cha86] for the standard case and in [JL93] for the generalized case. The basic idea is to construct a data structure $D(p)$ for each endpoint p to handle a special version of the segment intersection problem (called the *centered segment intersection* problem), in which the query segment has its supporting line passing through p .

We discuss in this paragraph how a general segment intersection query can be reduced to a constant number of specialized segment intersection problem mentioned above. The following discussion has already appeared in [Cha86] and [JL93], and is provided here only for completeness. As illustrated in Figure 1, given a general segment intersection query $q = \overline{ab}$, whose supporting line is denoted by K , we translate a copy of K in a direction perpendicular to K until an endpoint p of a segment in P is encountered, and denote the translated copy as H . Chazelle describes how to find p in $O(\log n)$ time using $O(n)$ space. We partition P into two subsets C and B with respect to p . The subset C consists of segments whose supporting lines pass through p , and B consists of all the other segments. The colors of the segments in C that intersect q can be reported in $O(\log n + k)$ time using $O(n)$ space (see [JL93] for details). Hence the remaining problem is to report the colors of the segments in B that intersect q . The lines H and K intersect exact the same sequence of segments s_1, s_2, \dots, s_k in P , in the same order. Using the techniques of Chazelle for handling the standard segment intersection problem, we can in $O(\log n)$ time to find the first segment

s_a that intersects \overline{ap} in the direction from a to p and the first segment s_b that intersects \overline{bp} in the direction from b to p . If both s_a and s_b intersect q , then the set of segments in B that intersect q is the union of the set of segments in B that intersect \overline{ap} and the set of segments in B that intersect \overline{bp} . If neither s_a nor s_b intersects q , then no segment in B intersects q and we return an empty set. Finally, in the case when one of s_a or s_b intersects q , without loss of generality, suppose s_b does. Let b' be the cross point of the segments \overline{bp} and s_b and c be the cross point of the segments \overline{bp} and s_a . We can in $O(\log n)$ time to find first segment s'_a in B that intersects \overline{cb} in the direction from c to b . Let a' be the cross point of the segments \overline{cb} and s'_a . The segments in P that intersect q are exactly those that intersect $\overline{a'b'}$. Therefore, we can in $O(\log n)$ time transform a general segment intersection problem into at most two centered segment intersection problems.

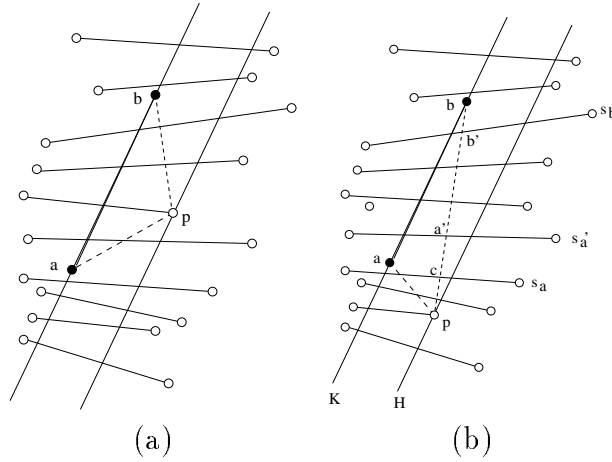


Figure 1: General segment intersection search

We now discuss how to use a partially persistent version of the data structure of Corollary 2.1 to handle a centered segment intersection query with respect to p . Consider a vertical ray shooting upwards from p . We assume that no segment in B crosses this ray. If this is not the case, we can simply cut along this ray each segment intersecting it into two pieces with the same color as the original segment. We first define a partial order \preceq on the segments in B . Consider two segments s_i and s_j . We say $s_i \preceq s_j$ if there exist a ray shooting from p that intersects s_i before s_j . Chazelle [Cha86] showed that there exists a total order \ll on B in which the partial order \preceq can be embedded such that for any two segments s and t , $s \preceq t \rightarrow s \ll t$. We assign each segment s in B an integer $r(s)$ which is the rank of s in B with respect to the total order \ll .

The data structure we use to handle the centered segment intersection problem is a persistent version of the data structure of Corollary 2.1, where the colored 1-D points maintained by this data structure are the ranks of the segments in B . Imagine that we rotate an upward-shooting ray R centered at p counter clockwise (we will denote the ray at its original position as R_0), as illustrated in Figure 2. Consider an endpoint of a segment s in B encountered in this process. If it is the first endpoint of s encountered, we insert $r(s)$ into the data structure with the version of this update operation being the angle α measured counter clockwise from R_0 to R . If the endpoint is the second one of s encountered, we delete

s from the data structure and assign a version to this update operation in the same fashion. By Corollary 2.1, the size of this partially persistent data structure is $O(n)$.

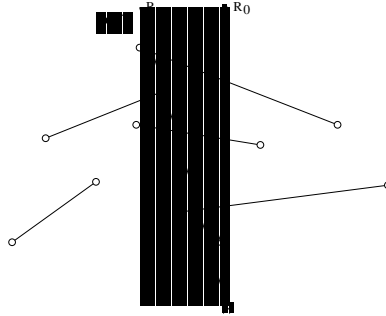


Figure 2: Building a persistent data structure using angular sweeping

Given a centered segment intersection query specified by the segment q , we can always assume that it is supported by a ray Q shooting from p . Otherwise q must pass through p and we can split it into two query segments, each supported by a ray shooting from p . Let β be the angle measured counter clockwise from R_0 to Q . We can find the appropriate version of the ephemeral 1-D data structure by looking for the version which is the predecessor of β . Searching this version takes $O(\log n)$ time.

Theorem 3. *There exists a data structure of size $O(n^2)$ for a set P of n colored segments in \mathcal{R}^2 such that a generalized segment intersection query on S can be handled in $O(\log n + k)$ time.*

5 2-D Orthogonal Range Search

Using Corollary 3.1 as a base and following the approach of Janardan and Lopez, we can handle the generalized 2-D orthogonal range searching problem on a set P of 2-D points in \mathcal{R}^2 in $O(\log n + k)$ time by storing $O(\log n)$ copies of each point, each in a data structure of Corollary 3.1.

What we do is to build a binary tree T on the y-coordinates of the points in P , each node v of T stores a key $k(v)$ such that the points stored in the subtree rooted at the left child u of v have their y-coordinates smaller than $k(v)$ and the points in the subtree rooted at the right child w of v have their y-coordinates larger than $k(v)$. The node v is associated with two secondary structures $L(v)$ and $R(v)$ which are appropriate versions of the data structure of Corollary 3.1. The structure $L(v)$ is built on the points stored in the subtree rooted at u and the structure $R(v)$ on the points stored in the subtree rooted at w . Given a query $(a, b; c, d)$, the highest node v whose key is contained in $[c, d]$ can be located in $O(\log n)$ time. The answer to the query then can be obtained by combining the results of two appropriate 3-sided 2-D queries on $L(v)$ and $R(v)$.

Theorem 4. *There exists a data structure of size $O(n \log n)$ for a set P of n colored 2-D points in \mathcal{R}^2 such that a generalized range query on P can be handled in $O(\log n + k)$ time.*

6 Rectangular Point Enclosure

Given a set P of colored rectangles in \mathcal{R}^2 whose edges are parallel to the axes, a generalized rectangular point enclosure problem is defined as finding the distinct colors of the rectangles in P that contain a query point $q = (a, b)$.

A generalized rectangle point enclosure problem can be viewed a “partially persistent” version of the generalized 1-D point enclosure problem. Suppose we already have a dynamic data structure D to handle the generalized 1-D point enclosure problem. Let P_y be the set of projections of the vertical edges of the rectangles in P to the y-axis. We use an imaginary horizontal line to sweep through these projections from top down. When we encounter the head of the projection of a rectangle $r = (x_1, x_2; y_1, y_2)$, we insert the interval (x_1, x_2) , which is assigned the same color as r , into D . When we encounter the tail of the same projection, we remove the interval (x_1, x_2) from D . Each insertion and deletion will be assigned a version ID which is the y-coordinate of the horizontal sweeping line at the time of that operation. A rectangular point query thus can be performed by first identifying the appropriate version of D and then using it to answer a generalized 1-D point enclosure query.

An optimal algorithm for handling 1-D generalized point enclosure query has been given by Jarnardan and Lopez [JL93]. As we did in dealing with the generalized 1-D range searching problem, we give an alternative solution which requires slightly more space and query time but supports update operations with a constant number of update steps.

we discuss the following *dynamic generalized 1-D point enclosure* problem.

Problem 6.1. *Let U be a list of n real numbers and let S be a set of $m \leq n$ colored 1-D intervals, the coordinates of whose endpoints are taken from U . Preprocess S so that (i) given a 1-D query point $q = (q.x)$, the distinct colors of the intervals $t = (t.x_1, t.x_2)$, $t.x_1 < t.x_2$ in S that satisfy $t.x_1 \leq q.x \leq t.x_2$ can be reported in $O(\log n + k)$ time; (ii) a new interval, the coordinates of whose endpoints are also from U , can be inserted into the existing data structure with $O(1)$ update steps; and (iii) an existing interval in S can be removed with $O(1)$ update steps.*

Following the approach of [JL93], we first construct a set S' of intervals by merging overlapping intervals in S with the same color into a single interval. It is obvious that no two intervals in S' with the same color overlap. Therefore a generalized 1-D point enclosure query on S' is equivalent to a standard 1-D point enclosure query. Instead of using the *window list* technique of Chazelle [Cha86], as Jarnardan and Lopez did, to handle this query, we use the interval tree [Ede80]. We build a binary tree T on the set of sorted real numbers in U by recursively choosing a key $k(v)$ for each internal node v such that half the numbers stored in T_v are less than $k(v)$ and the other half are greater than $x(v)$. For each interval t in S' , let v be the highest node such that the key $k(v)$ is contained within t . Notice that each interval will be associated with exactly one internal node. At each node v , we store two lists $IL(v)$ and $IR(v)$ that contain separate copies of the intervals associated with v . Both $IL(v)$ and $IR(v)$ are linked lists. The intervals in $IL(v)$ are sorted by the increasing order of the x-coordinates of their left endpoints and the intervals in $IR(v)$ are sorted by the decreasing order of the x-coordinates of their right endpoints.

A query can be handled by visiting one of the two sorted lists associated with each of the internal nodes on the path from the root of T to the leaf node corresponding to the

predecessor of $q.x$. For example, if, at node v , $q.x < k(v)$, then we only need to scan the list $IL(v)$ until we encounter an interval whose left endpoint is to the right of $q.x$. The query time is obviously $O(\log n + k)$.

Now consider the update operations. First consider adding a new interval t of color c . The addition of this new interval causes S' to be updated. Consider the subset of S' consisting of the intervals with color c . Since no two intervals from this subset overlap, the introduction of interval t may cause (i) an existing interval to be extended, (ii) two existing intervals to be merged with t , or (iii) a simple addition of t to S' . In any of the three cases, we update S' by removing at most two intervals and adding at most one interval. Next consider deleting an existing interval t . Again consider $S'(v)$. There is exactly one interval in $S'(v)$ that contain t . The deletion of t may cause an existing interval to (i) shrink, (ii) split into two, or (ii) be removed entirely. In any of these cases, S' can be updated with at most one deletion and two insertion operations. Therefore, a constant number of addition and deletion operations on the interval tree are sufficient for each addition or deletion of a colored interval, and hence we only need to analyze the complexity of insertion and deletion operations on the interval tree. It is obvious that a new interval only need to be associated with one internal node v in T and hence inserted into $IL(v)$ and $IR(v)$. To remove an existing interval we only need to update two sorted lists as well. Both insertion and deletion operations on a sorted list clearly can be performed using a constant number of update steps.

By making the data structure described above partially persistent, we obtain a linear space data structure for handling the rectangular point enclosure problem.

Theorem 5. *There exists a data structure of size $O(n)$ for a set P of n colored 2-D axis-parallel rectangles in \mathcal{R}^2 such that given a 2-D query point q , the colors of the rectangles in S that contain q can be reported in $O(\log n + k)$ time.*

Combining this result with the techniques for extending low dimensional results to higher dimensions [JL93, AGM02], we immediately obtain better results for higher dimensional point enclosure problems.

7 Rectangle Intersection Search

As Edelsbrunner and Overmars have observed in [EO82], the answer to a standard rectangle intersection search problem can be obtained by combining the answers to a constant number of orthogonal segment intersection, rectangular point enclosure, and 2-D orthogonal range searching queries. Combining Theorems 2, 4, and 5, we obtain:

Theorem 6. *There exists a data structure of size $O(n \log n)$ for a set P of n colored 2-D axis-parallel rectangles in \mathcal{R}^2 such that given a 2-D query rectangle q , the distinct colors of the rectangles in S that intersect q can be reported in $O(\log n + k)$ time.*

8 Conclusion and Future Directions

Using the notion of persistent data structures and incorporating a number of transformation techniques, we were able to develop algorithms for a set of generalized intersection problems

to achieve performance results that match or almost match the lower bounds of the corresponding standard intersection searching problems. Our results provide additional support to the conjecture that generalized intersection searching problems are no more difficult than their standard counterparts for low-dimensional spaces (at least for $d \leq 2$).

In addition to the problems mentioned in [GJS95] and [GJS], the following two problems seem to deserve further attention:

- (i) Does there exist a $O(n \log n / \log \log n)$ space and $O(\log n + k)$ query time algorithm for the generalized 2-D range searching problem under the pointer machine model?
- (ii) Is there a general technique that transforms an efficient solution to a standard intersection problem into a solution with similar complexity bounds to the corresponding generalized intersection problem?

References

- [AGM02] Pankaj K. Agarwal, Sathish Govindarajan, and S. Muthukrishnan. Range search in categorical data: color range searching on grid. In *Proceedings of the 10th European Sympos. Algorithms*, pages 17–28, 2002.
- [AvKO93] P.K. Agarwal, M. van Kreveld, and M. Overmars. Intersection queries for curved objects. *Journal of Algorithms*, 15:229–266, 1993.
- [BM95] A. Boroujerdi and B.M.E. Moret. Persistency in computational geometry. In *Proc. 7th Canadian Conf. Comp. Geometry (CCCG 95)*, pages 241–246, Qubec, Canada, 1995.
- [Cha86] Bernard Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal on Computing*, 15(3):703–724, August 1986.
- [Cha90] Bernard Chazelle. Lower bounds for orthogonal range search I. The reporting case. *Journal of the ACM*, 37(2):200–212, 1990.
- [CL01] Bernard Chazelle and Ding Liu. Lower bounds for intersection searching and fractional cascading in higher dimension. In *Proceedings of the 33th Annual ACM Symposium on Theory of Computing*, pages 322–329, 2001.
- [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.
- [Ede80] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical Report Rep. F59, Tech. Univ. Graz, Institute für Informationsverarbeitung, 1980.
- [EO82] H. Edelsbrunner and M.H. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters*, 14:124–127, 1982.

- [GJS] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Computational geometry: generalized intersection searching. In Dinesh Mehta and Sartaj Sahni, editors, *Handbook of Data Structures and Applications*. CRC Press. To appear.
- [GJS94] Prosenjit Gupta, Ravi Janardan, and Michiel H. M. Smid. Efficient algorithms for generalized intersection searching on non-iso-oriented objects. In *Symposium on Computational Geometry*, pages 369–378, 1994.
- [GJS95] Prosenjit Gupta, Ravi Janardan, and Michiel Smid. Further results on generalized intersection searching problems: counting, reporting, and dynamization. *Journal of Algorithms*, 19:282–317, 1995.
- [JL93] Ravi Janardan and Mario Lopez. Generalized intersection searching problems. *International Journal of Computational Geometry & Applications*, 3(1):39–69, 1993.
- [Mor03] Christian Worm Mortensen. Generalized static orthogonal range searching in less space. Technical Report TR-2003-22, The IT University of Copenhagen, 2003.
- [Mut02] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc. 13th Annual Symposium on Discrete Algorithms*, pages 657–666, 2002.