Persistence in Computational Geometry[1]

Ali R. Boroujerdi
Naval Research Laboratory
4555 Overlook Avenue S.W.
Washington, DC 20375

Bernard M.E. Moret
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131-1386

ABSTRACT

We show how persistence can be used to solve a number of geometric problems where preprocessing is required to facilitate query answering. Efficient solutions for most of the problems discussed already exist in the literature; however, persistence provides an efficient and conceptually simpler alternative to existing solutions.

1 Introduction

Suppose we have a collection of geometric objects and are asked to report those satisfying a given property with respect to a query object (not necessarily part of the collection). If many such queries are anticipated, we want to preprocess the collection and build a data structure to enable us to answer queries efficiently. The efficiency of such solution is then measured by three parameters: the preprocessing time, the space required to store the data structure, and the time required to answer a query—with preprocessing time generally considered unimportant. We refer to an algorithm with $O(f(n))$ storage requirement and $O(g(n) + k)$ query time, where k is the number of objects reported, as an $(f(n), g(n))$-algorithm.

In this paper, we show how persistence can be used to solve a number of query-type problems in computational geometry: interval intersection, segment intersection, point enclosure, grounded 2D search, orthogonal range search, and line of sight. Efficient solutions for most of these problems already exist in the literature; persistence provides an efficient and conceptually simpler alternative to existing solutions. The idea of building a persistent data structure to answer geometric queries is not new. Sarnak and Tarjan [9] first showed how to use persistence to solve a geometric problem (planar point location) for which existing solutions were all fairly complex; they used a persistent red-black tree to provide an optimal $(n, \log n)$ solution. (For details on persistence and various methods of making data structures persistent, the reader is referred to the comprehensive paper of Driscoll *et al.* [4].) What makes persistence so well-suited for geometric problems is the prevalence of sweep methods in computational geometry and the ease with which persistence can be used with most sweep methods.

2 Interval Intersection

Given a set, S, of n intervals $\{[a_1, b_1], [a_2, b_2], \dots, [a_n, b_n]\}$ on the real line, the *interval intersection* (or interval overlap) problem is to report all the intervals in S that intersect a query interval $[x, y]$. (Two intervals intersect if and only if they share at least one point.) Optimal $(n, \log n)$-solutions include the *interval tree* of Edelsbrunner [5], the *priority search tree* of McCreight [7], and the *window-list* of Chazelle [3]. For the discrete version of the problem (*i.e.* when the endpoints of the query intervals are taken from a range of $O(n)$ integers), the window-list yields a $(n, 1)$-algorithm, until now the only optimal algorithm known for the discrete version of the problem.

We describe a persistent data structure, which we call the *interval list*, that provides an efficient alternative to the existing solutions; this structure is basically a partially persistent linked list.[2] The

[1]This work is supported by the Office of Naval Research under contract N00014-92-C -2144.

[2]A partially persistent structure can be viewed as having a version for each point on the real line; in general, if version numbers are drawn from a totally ordered set, the structure can be viewed as having a version for each element of the set.

version corresponding to point x consists of the values with the largest version stamp not exceeding x. Given a set, S, of n intervals $\{[a_1, b_1], [a_2, b_2], \ldots, [a_n, b_n]\}$, we build the corresponding interval list, L, so that version x of L consists of the intervals in S that contain point x. Thus versions x_1 through x_2 consist of the intervals that intersect $[x_1, x_2]$. L is constructed by making a left-to-right sweep of the real line. Before starting the sweep, we create an empty version of L, stamped $-\infty$. As the sweep encounters a point, x, marking the beginning of one or more intervals, we create a new version, x, by appending the interval(s) onto L using version stamp x. As the sweep encounters a point, y, marking the end of one or more intervals, we create a new version, $y + \varepsilon$, by deleting the interval(s) from L using version stamp $y + \varepsilon$, where ε is small enough so that any point to the right of y does not fall to the left of $y + \varepsilon$. If the sweep encounters a point, z, that marks the beginning of one or more intervals as well as the end of one or more (possibly different) intervals, we of course create version z before version $z + \varepsilon$. The access pointers (*i.e.* pointers to the heads of the different versions of L) are stored in an array, so that a binary search can locate the access pointer to a given version.

Sorting the endpoints of the intervals in S takes $O(n \log n)$ time. Constructing L requires making n insertions and n deletions; each involving a constant number of update steps. Thus, using the node-copying method, which has an amortized time cost of $O(1)$ per update step, an insertion or deletion can be performed at an amortized time cost of $O(1)$ provided we have constant-time access to the tail of the newest version of L (needed for insertion) and to an interval's node in the newest version of L through its right endpoint (needed for deletion). Thus the entire sequence of insertions and deletions takes $O(n)$ time, resulting in $O(n \log n)$ preprocessing time. The node-copying method has an amortized space cost of $O(1)$, hence L requires $O(n)$ storage.

To report the intervals containing point x, we simply retrieve version x of L. This takes $O(\log n)$ time to locate the access pointer plus $O(k)$ time to report the intervals, where k is the number of intervals in version x of L, resulting in a total retrieval time of $O(\log n + k)$. To report the set of intervals intersecting $[x_1, x_2]$, we note that if an interval intersects $[x_1, x_2]$, then it either contains x_1 or its left endpoint is contained in $(x_1, x_2]$. Let S_1 be the set of intervals containing x_1 and S_2 be the set of intervals whose left endpoints are in $(x_1, x_2]$. The intervals in S_1 are in version x_1 of L and can be reported in $O(\log n + |S_1|)$ time. Sorting the left endpoints of the intervals in S and storing the resulting sequence in an array enables us to report the intervals in S_2 in $O(\log n + |S_2|)$ time. Thus queries can be answered in $O(\log n + k)$ time.

For the discrete version of the problem, we can normalize the set of m query endpoints to integers in the range $[1, m]$ and create an array of access pointers with an entry for each query endpoint. Access pointers can then be retrieved in constant time. If an array of left endpoints is used to answer queries, this array too must be modified to allow constant-time access to its relevant entries. Queries can then be answered in $O(k)$ time.

3 Segment Intersection

Given a set, S, of n line segments in the plane, the *segment intersection* problem is to report the segments in S that intersect a query segment q. Optimal $(n, \log n)$-solutions for three restricted versions of the problem have been given by Chazelle [3]. Here, we use persistence to obtain other optimal solutions.

First, we consider orthogonal segment intersection, *i.e.* when query and data set segments are mutually orthogonal. Let S be a set of n segments in the plane, with the segments in S horizontal and the query segments vertical. We build a persistent data structure, T, such that version i of T consists of the segments in S that intersect the line $x = i$. To support efficient search, T is built as a persistent search tree. The segments in each version of T are ordered with respect to their y-coordinates. Let (x, y_1) and (x, y_2) be the endpoints of a query segment q. Version x of L contains all the segments that could possibly intersect q. The segments that actually intersect q can be located by a range search on this version, using y_1 and y_2 as the keys. The access pointers are stored in an array. The access pointer to version x can then be located using a binary search, taking $O(\log n)$ time. The segments in version x with y-coordinates in the range $[y_1, y_2]$ can be reported in $O(\log n + k)$ time, where k is the number of segments reported. Queries can thus be answered in $O(\log n + k)$ time.

T is constructed by making a left-to-right sweep of the plane with a vertical line. As the sweep line encounters the beginning (respectively the end) of one or more line segments at x-coordinate i, the segments are inserted into (respectively deleted from) T using version stamp i (respectively $i+\varepsilon$), where ε is small enough so that any point to the right of i does not fall to the left of $i+\varepsilon$. Thus T is constructed by making a total of n insertions and n deletions. Sorting the endpoints of the segments by x-coordinate takes $O(n \log n)$ time. An update operation in a balanced search tree of size n requires $O(\log n)$ time. The node-copying method makes a linked structure persistent at a worst-case time cost of $O(1)$ per access step and an amortized time cost of $O(1)$ per update step. Thus the entire sequence of insertions and deletions can be performed in $O(n \log n)$ time, resulting in $O(n \log n)$ preprocessing. The storage requirement depends on the number of update steps per insertion/deletion. By keeping the number of update steps per insertion/deletion down to a constant (even in the amortized sense), we obtain $O(n)$ storage. This can be done with a red-black binary search tree, since insertions and deletions in these trees require only $O(1)$ pointer changes (see [8]); only recolorings may propagate for more than $O(1)$ stages, but, since we only update the newest version of the tree and since color fields are only used in update operations, the color fields can be overwritten and thus do not require extra storage.

Next, we consider the problem where query segments have their supporting lines passing through a fixed point p. In this case, we build a persistent search tree, T, by making an angular sweep of the plane with a line passing through p. Since an angular sweep does not have a natural starting point, the initial position of the sweep line is arbitrary. Version 0 of the search tree consists of the segments that intersect the sweep line at its initial position. We then start rotating the sweep line 180 degrees. As the sweep line, having rotated α degrees, encounters the beginning (respectively the end) of one or more line segments, the segments are inserted into (respectively deleted from) T using version stamp α (respectively $\alpha+\varepsilon$), where ε is small enough so that any angle larger than α is at least as large as $\alpha+\varepsilon$. The segments in each version of T are ordered with respect to the signed distance from p of their intersection point with the sweep line. Assuming input segments can only intersect at their endpoints, the relative order of segments does not change from one version to the next. The structure requires $O(n)$ space and $O(n \log n)$ preprocessing time. Let the angle between the initial position of the sweep line and the supporting line of a query segment, q, be α degrees. To report the segments intersecting q, we search version α of T, looking for segments that intersect the supporting line of q between q's endpoints. This can be accomplished by a range search. Queries can thus be answered in $O(\log n + k)$ time, where k is the number of segments reported.

Finally, we consider the problem where query segments have a fixed slope. In this case, we build a persistent search tree, T, by making a sweep of the plane with a line that has the same slope as query segments. Assume that query segments are parallel to the y-axis. Version i of the search tree consists of the segments that intersect the line $x = i$. The segments are ordered by the y-coordinate of their intersection point with this line. Again, note that the relative order of the segments does not change from one version to the next if input segments only intersect at their endpoints. The structure requires $O(n)$ storage and $O(n \log n)$ preprocessing time. Queries can be answered in $O(\log n + k)$ time, where k is the number of segments reported.

4 Point Enclosure

Given a set, S, of n d-ranges in d-dimensional space, the *d-dimensional point enclosure* problem is that of reporting the d-ranges (rectangular boxes parallel to the axes) of S that contain a query point q. A number of solutions to the problem have appeared [3, 7, 10]. In particular, Chazelle [3] proposed an optimal $(n, \log n)$-algorithm for the 2-dimensional version of the problem which generalizes to a $(n \log^{d-2} n, \log^{d-1} n)$-algorithm in $d(>1)$ dimensions. Here, we use persistence to obtain a simpler optimal solution for the 2-dimensional version of the problem, which also generalizes to a $(n \log^{d-2} n, \log^{d-1} n)$-solution in $d(>1)$ dimensions.

Given a set, S, of n intervals on the real line, the interval tree of Edelsbrunner [5] reports the intervals in S that intersect a query interval $[x_1, x_2]$ in $O(\log n)$ time. The structure requires $O(n)$ storage and

$O(n \log n)$ preprocessing time. A solution to the 2-dimensional point enclosure problem with the desired space and time bounds could be obtained using a persistent interval tree, but this would be overkill: all we need is a data structure that reports those intervals in S which contain a query point x. This is the 1-dimensional point enclosure problem and a special case of the interval intersection problem where the query interval is a single point.

For a sequence of points $(x_1, x_2, \ldots, x_n)$ and a set of intervals S with endpoints from the sequence, our simplified interval tree, T, is recursively defined as follows. The primary structure of T is a balanced binary tree. The root, w, has a discriminant $\delta(w) = (x_{\lfloor \frac{n}{2} \rfloor} + x_{\lfloor \frac{n}{2} \rfloor + 1})/2$ and points to two secondary structures, $L(w)$ and $R(w)$. $L(w)$ contains the sorted list (in ascending order) of the left endpoints of the intervals in S containing $\delta(w)$. $R(w)$ contains the sorted list (in descending order) of the right endpoints of the intervals in S containing $\delta(w)$. The left subtree of w is the interval tree for the sequence of points $(x_1, x_2, \ldots, x_{\lfloor \frac{n}{2} \rfloor})$ and the subset S_L of S containing the intervals whose right endpoints are to the left of $\delta(w)$. The right subtree of w is defined analogously. The tree for a sequence containing a single point, x_i, is a single node with a discriminant equal to x_i. To support efficient insertion and deletion of intervals, the secondary structures are stored as threaded balanced search trees with two access pointers, one to the root and the other to the smallest (largest) element in the list.

Insertion of an interval $[b, e]$ consists of finding the highest node, v, of T such that $b \leq \delta(v) \leq e$,[3] followed by the insertion of b into $L(v)$ and e into $R(v)$. To report the set of intervals containing point x, we trace a path from the root to a leaf of T using x as the key—unless we encounter a vertex, v, along the path such that $\delta(v) = x$, in which case the path terminates at v. If $x < \delta(v)$, we scan $L(v)$ in ascending order reporting every interval whose left endpoint is less than x, something we can do in $O(k)$ time. If $x > \delta(v)$, we scan $R(v)$ in descending order reporting every interval whose right endpoint is greater than x. If $x = \delta(v)$, a scan of $L(v)$ (or alternatively $R(v)$) in its entirety will produce the intervals containing x as yet unreported. The structure requires $O(n)$ storage and $O(n \log n)$ preprocessing time. Queries can be answered in $O(\log n + k)$ time.

The structure described is a linked structure with nodes of constant bounded in-degree. It can thus be made persistent, using the node-copying method, at a worst-case time cost of $O(1)$ per access step and an amortized time and space cost of $O(1)$ per update step. The 2-dimensional point enclosure problem can be solved by building a persistent form of the structure as follows. Given a set, S, of n orthogonal objects (*i.e.* rectangles with sides parallel to the axes) in 2-dimensional space, create an empty version, stamped $-\infty$, of the structure and make a left to right sweep of the plane with a vertical line. As the sweep line encounters one or more rectangles at x-coordinate i, create a new version, i, of the structure and insert the y-intervals of the rectangles into the newly created version. As the sweep line encounters the end of one or more rectangles at x-coordinate j, create a new version, $j + \varepsilon$, of the structure and delete the y-intervals of the rectangles from the newly created version. The net effect is that version i of the structure contains the y-intervals of the rectangles whose x-intervals contain point i. Given a query point (x, y), we simply search version x of the structure and report the rectangles associated with the y-intervals in version x containing point y. Implementing the secondary structures as red-black trees, the entire structure requires $O(n)$ storage and $O(n \log n)$ preprocessing time. Queries can be answered in $O(\log n + k)$ time. The algorithm can be generalized to an $(n \log^{d-2} n, \log^{d-1} n)$-algorithm in d dimensions using the segment tree of Bentley [2].

5 Grounded 2-Dimensional Range Searching

Given a set, S, of n points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ and a query triple (a, b, c) with $a \leq b$, the problem is to report the set of points $\{(x_i, y_i) : a \leq x_i \leq b \textit{ and } y_i \leq c\}$. The *priority search tree* of McCreight [7] yields an optimal $(n, \log n)$-solution to this problem. Persistence, however, offers an alternative to the priority search tree. We construct a persistent search tree, T, such that version i of T contains every point (x_j, y_j) for which $y_j \leq i$, by making a bottom-to-top sweep of the plane with a horizontal line. The points are stored in the tree using their x-coordinates as keys. Answering a query

[3] Note that equality holds only when v is a leaf, in which case the interval being inserted is a single point.

given by the triple (a, b, c) consists of reporting all points in version c of T whose x-coordinates are in the range $[a, b]$. Queries can thus be answered in $O(\log n + k)$ time. Since the construction of T involves n insertions and no deletions, implementing it as a red-black tree (for example) takes $O(n)$ storage. Thus the solution is optimal. Unlike the priority search tree, this data structure is static—points cannot be inserted into the structure or deleted from it; on the other hand, it can easily accommodate duplicate x-coordinates, something the priority search tree cannot do.

6 Orthogonal Range Searching

Let S be a set of points in d-dimensional space. The *d-dimensional orthogonal range searching* problem is that of reporting all the points in S that are contained in a query d-range R. Much attention has been given to this problem. The *range tree method* (see [1]) yields a data structure that allows queries to be answered in $O(\log^d n + k)$ time using $O(n \log^{d-1} n)$ space and preprocessing time. Willard [11] and Lueker [6] devised a *layering* technique (also know as *fractional cascading*) to reduce the query time to $O(\log^{d-1} n + k)$ without affecting the storage requirement or preprocessing time. We show that persistence can be used in orthogonal range searching to achieve the same effect as fractional cascading.

Note that the d-dimensional range tree is a linked data structure with nodes of constant bounded in-degree, so that we can make the structure persistent at a worst-case time cost of $O(1)$ per access step and an amortized time and space cost of $O(1)$ per update step by using the node-copying method. We solve an instance of the d-dimensional range search problem by reducing it to two instances of the persistent $(d-1)$-dimensional problem. Let S be a set of n points in d-dimensional space. We construct a complete binary tree, T, with n leaves. Each leaf of T is associated with a point, p, in S and has a key equal to the first coordinate of p. The internal nodes of T are assigned keys consistent with the rules of binary search trees. (Since we allow duplicates in T, the left or right subtree of a node, v, may contain a key equal to that of v.) Let v_L and v_R be the left and right children of node v respectively and $P(v)$ be the set of points associated with the leaves of the subtree rooted at v. With each internal node, v, of T, we associate two persistent structures, $D_L(v)$ and $D_R(v)$, corresponding to the solutions of two instances of the persistent $(d-1)$-dimensional range search problem. Version i (respectively j) of $D_L(v)$ (respectively $D_R(v)$) contains those points in $P(v_L)$ (respectively $P(v_R)$) whose first coordinates are in the range $[i, key(v)]$ (respectively $[key(v), j]$). $D_L(v)$ is constructed starting with the highest version. In retrieving version i of $D_L(v)$, we follow the pointers with the least version number no less than i. Each leaf, v, of T is also associated with two persistent structures. However, in this case one of the structures is empty and the other contains the single point in $P(v)$. We identify one of the structures as $D_L(v)$ and the other as $D_R(v)$. Since each point in S can appear at most once at each level of T, the structure requires $O(n \log^{d-1} n)$ space and preprocessing time.

To report the set of points in a d-range $R = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_d, b_d]$, we find the highest node v of T whose key is in the range $[a_1, b_1]$. Version a_1 (respectively b_1) of $D_L(v)$ (respectively $D_R(v)$) contains the points whose first coordinates are in the range $[a_1, key(v)]$ (respectively $[key(v), b_1]$). (Each point with first coordinate equal to $key(v)$ is either in version a_1 of $D_L(v)$ or version b_1 of $D_R(v)$, but not both.) We now search version a_1 (respectively b_1) of $D_L(v)$ (respectively $D_R(v)$) and report the points in the $(d-1)$-range $[a_2, b_2] \times \cdots \times [a_d, b_d]$. The time required to answer a query is $O(\log n + T)$, where T is the time it takes to solve two instances of the persistent $(d-1)$-dimensional range search problem. Queries can thus be answered in $O(\log^{d-1} n + k)$ time, where k is the number of points reported.

7 The Line-of-Sight Problem

Given a point, p, and a set of polyhedral objects, S, in d-dimensional space, the *line-of-sight* problem is that of determining whether a query point, q, is visible from point p. We provide a solution for the 3-dimensional version of the problem where S is a collection of simple polygons in 3-dimensional space by reducing the problem to one of planar point location and using the solution of Sarnak and Tarjan [9].

Let each point, q, be represented by its polar coordinates with origin at p: a radius, r, which is the length of the segment, s, connecting q to the origin; an angle, θ, which is the angle between s and the xz-plane; and an angle, ϕ, which is the angle between s and the xy-plane. We remove all surfaces hidden from p and partition the remaining ones into convex pieces. (Hidden surface removal and partitioning simple polygons into convex pieces are well understood problems in computational geometry.) Now we (conceptually) project all visible convex pieces onto a half-sphere centered at p—effectively ignoring the radius coordinate. Given a query point q, we begin by locating its projection onto the half-sphere; this is essentially the same problem as planar point location, since the half-sphere is topologically equivalent to a plane and is partitioned into convex regions. We use the optimal $(n, \log n)$ algorithm of Sarnak and Tarjan with suitable modifications to take into account the circularity of the setting. Once we have located the convex polygon P in which the projection of q sits, we decide in constant time whether q is visible from p by checking whether q is in front of P or behind it.

Since the number of visible convex pieces remains linear in the size of the input (the description of the polyhedra forming the scene), we have solved the static line-of-sight problem with an optimal $(n, \log n)$ algorithm based on a persistent red-black tree.

8 Conclusion

The use of persistence in computational geometry yields conceptually simple and often optimal static solutions. Dynamic solutions, however, are beyond the scope of the persistent structures currently known. Finding an effective way to allow update operations that change several versions simultaneously, would result in efficient dynamic solutions; however, in several cases, these solutions would be so efficient as to violate lower bounds for comparison-based methods—a possible indication of what remains doable with persistent data structures.

References

[1] J. L. Bentley. Decomposable searching problems. *Info. Proc. Lett.*, 8:244–251, 1979.

[2] J. L. Bentley and D. Wood. An optimal worst-case algorithm for reporting intersections of rectangles. *IEEE Trans. Comput.*, C-29:571–577, 1980.

[3] B. Chazelle. Filtering search: A new approach to query answering. *SIAM J. Comput.*, 15:703–724, 1986.

[4] 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.

[5] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical Report F59, Tech. Univ. Graz, 1980.

[6] G. S. Lueker. A data structure for orthogonal range queries. In *Proceedings, 19th Annual IEEE Symp. on Foundations of Computer Science.*, pages 28–34, 1978.

[7] E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14:257–276, 1985.

[8] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP*, volume 1. Benjamin/Cummings, 1991.

[9] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Comm. ACM*, 29:669–679, 1986.

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

[11] D. E. Willard. New data structures for orthogonal range queries. *SIAM J. Comput.*, 14:232–253, 1985.