Efficient Algorithms for Reporting Intersections

Garret Swart
Richard Ladner

Computer Science Department
University of Washington
Seattle, WA 98195
Technical Report No. 83-07-03

Abstract

The problem of taking a set of geometric objects in the plane and reporting all the pairwise intersections between the objects is considered. We introduce two new algorithms with a new supporting data structure for solving this problem for a wide class of geometric objects. These algorithms illustrate a new trade off in the design of plane sweep algorithms.

This research is supported in part by the National Science Foundation Grant MCS-80-03337.

1. Introduction

The *Intersection Reporting Problem* is that of taking a set of geometric objects in the plane and reporting all the pairwise intersections between the objects. This problem was first proposed by Shamos and Hoey [10] and has applications as diverse as computer graphics, integrated circuit design and video games. Solutions to the problem have been given for the special cases where the geometric objects are line segments (Bentley and Ottman [1]) and aligned rectangles (Bentley and Wood [2] and McCreight [8]). This work has been extended to the online reporting of intersecting pairs of rectilinear line segments (Vaishnavi and Wood [12] and to aligned rectangles in higher dimensions (Six and Wood [11]). Related developments in the solution of intersection problems include detecting whether two convex polygons or polyhedra intersect (Chazelle and Dobkin [3] and Dobkin and Kirkpatrick [4]), determining the planar regions formed by a self intersecting polygon and finding the intersection of two convex planar maps (Nievergelt and Preparata [9]).

In this paper we present two new algorithms, the *Major Chord Algorithm* and the *Chain Algorithm*, for reporting all pairs of intersecting planar geometric objects. Both algorithms in their basic forms report intersections with possible repetitions but modifications are presented to eliminate multiple reports. The first algorithm requires the objects be convex while the second requires only that the objects be simple. A planar object is *simple* if it is topologically equivalent to a disk. Figure 1 illustrates some simple and non-simple objects.

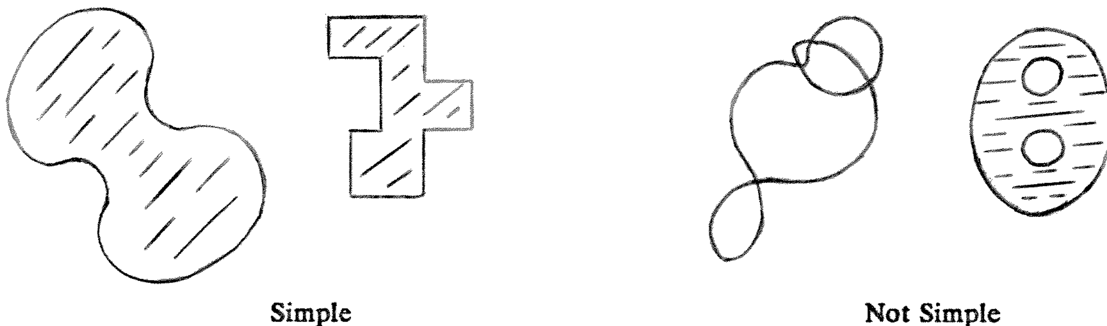


Figure 1: Examples of simple objects

The Intersection Detection Problem, the problem of detecting if two or more objects in a set intersect, is trivially reducible to the Intersection Reporting Problem. The Intersection Detection Problem for n objects is widely believed to require $\Omega(n \log n)$ time, and in fact does require it on a linear decision tree, based on a reduction from point uniqueness and a result of Dobkin and Lipton [5]. However, the relevance of this result is in question when the objects are complex as line segments because the problem cannot be solved using linear tests. Another constraint on the running time of an algorithm solving this problem is the size of the output, the number of pairs of objects which intersect, which we will denote by s. Thus, the best one can reasonably hope for in a solution

to this problem is $O(n \log n + s)$. Though the algorithms presented here do not meet this goal in general, they have cases where they do meet it or come very close.

Both algorithms share a number of features including the fact that both are plane sweep algorithms and both employ a new data structure for maintaining the information needed during the plane sweep. The new data structure, called MV-trees (short for multiple version trees), may have other applications.

Major Chords

Both algorithms take different views of the objects to be processed and gain in efficiency if the objects 'nice' with respect to those views. The Major Chord Algorithm views each convex object together with its *major chord*, the line segment from its left-most, top-most point to its right-most, bottom-most point. Figure 2 illustrates a convex object together with its major chord.

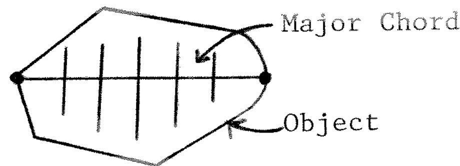


Figure 2: Example of a convex object with its major chord

Typical convex objects we are considering include convex polygons, ellipses, and composite objects whose boundary is made up of line segments and arcs. For the algorithm to be well defined we must be able to (i) compute the intersection of an object and a line and (ii) detect the intersection of two objects. For our computations of running times we will assume that the objects are convex polygons, in which case the computations (i) and (ii) can be done in time $O(\log k)$, where k is the number of line segments in the polygon. Chazelle and Dobkin [3] and Dobkin and Kirkpatrick [4] present algorithms which solve these problems for convex polygons; they may be modified to deal with convex composite objects. Table 1 summarizes the running times of the Major Chord Algorithm.

The trivial algorithm of checking all possible pairs for intersection runs in time $O(n^2 \log k)$ which is better than the Major Chord Algorithm when the depth of intersection is $\Theta(n)$. However, if the depth of intersection is small then the new algorithm performs much better than the trivial one. The algorithm's performance meets the asymptotically optimal bound of $O(n \log n + s)$ in the case where both d and k are bounded by a constant.

In different applications the depth of intersection is bounded. For example, a VLSI layout has a small fixed number of layers in which components may overlap without an error occurring, so the maximum depth of intersection in a error free layout is constant. A VLSI design rule checker could

Major Chord Algorithm	$O(n\, d^2 \log k + n\, d \log n + s\, d \log k)$
Best case $d = O(1)$	$O((n + s)\log k + n \log n)$
Worst case $d = \Theta(n)$, $s = \Theta(n^2)$	$O(n^3 \log k)$

where,
n = number of polygons
k = number of line segments per polygon
s = number of polygon-polygon intersections (the things we are interested in!)
d = maximum depth of intersection, maximum number of polygons containing any one point on the plane.

Table 1: Running Times of the Major Chord Algorithm

be designed to check that intersections between objects occur only between the expected pairs but even an erred layout would not be expected to have a depth of intersection more than a constant factor greater than the number of layers. In a video game the general situation may have no objects intersecting and an intersection may indicate that the player has just crashed and that the game is over.

Chains

The Chain Algorithm views its simple, possibly non-convex, objects as bounded by a set of *chains*, where each chain is a connected sequence of *segments* ordered by increasing x-coordinate. A segment is a non intersecting curve which intersects any vertical line at most once (i. e. the intersection must be connected) and intersects any other segment at most a constant number of times. Segments considered include line segments and arcs of ellipses. Figure 3 illustrates the decomposition of a simple object into chains.

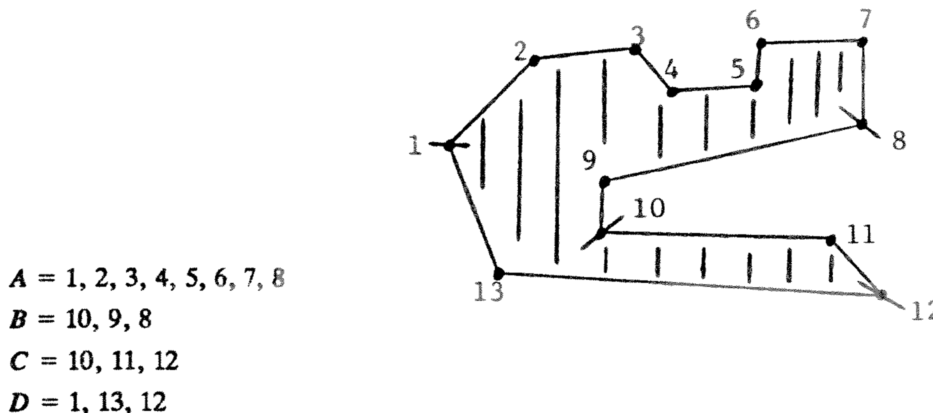


Figure 3: Example of a simple object decomposed into chains

For the Chain Algorithm to be well defined we must be able to compute (i) the intersection of two segments and (ii) compute the intersection of a segment and a line. In the cases we will consider these operations can be computed in constant time, as they can in the case of simple polygons. Table 2

Chain Algorithm	$O((r + n\,k)(\log k + \log n) + s)$
Best case $k = O(1)$	$O((s + n) \log n)$
Worst case $r = \Theta(n^2 k^2)$	$O(n^2 k^2 (\log k + \log n))$

where,
n = number of chains
k = number of segments per chain
s = number of object-object intersections (the things we are interested in!)
r = number of segment-segment intersections.

Table 2: Running Times of the Chain Algorithm

summarizes the running times for the Chain Algorithm for simple polygons. A trivial algorithm for checking all possible object-object intersections runs in time $O(n^2\, k \log k)$ so if the number of actual segment-segment intersections is $\Theta(n^2 k^2)$ then the Chain Algorithm is worse than the trivial algorithm. For many cases the Chain Algorithm will perform much better. In the case of objects of bounded size the algorithm runs in time very close to optimal. In many situations in vector computer graphics the number of segments making up any one object is bounded and the Chain Algorithm runs in time $O((s + n) \log n)$.

2. Common Features of Both Algorithms

Both the major chord and chain algorithms share the *plane sweep* algorithmic structure and share the use of the underlying MV-tree data structure.

Plane Sweeps

A plane sweep algorithm proceeds by conceptually sweeping a vertical line from left to right, keeping track of the input objects as they intersect the sweep line and solving the problem as the line sweeps forward. There are two structures common to all plane sweep algorithms, a *To-Do-List* of work yet to be done and a *y-structure* representing the one dimensional information needed to be maintained about the intersections of the objects with the current sweep line. The To-Do-List is a priority queue of activities, ordered by the x-coordinates of the points where the activity is to take place. The priority queue, in effect, implements the sweep line by the correspondence that the x-coordinate of the top element of the queue is the current position of the sweep line. As new activities to be done are found they are inserted into the queue. New things to be done are always found to the right of the current sweep line, thus the sweep line never backs up. The To-Do-List has operations: 'insert(thing)', where 'thing' is an activity to be done, and 'thing ← top', where 'thing' becomes the next activity to be done.

Typically, the y-structure represents some of the information about the intersection of the geometric objects with the sweep line. As such it must change dynamically as the sweep line is advanced. It also must represent the objects in such a way that the problem restricted to the sweep line may be solved. The activities placed on the To-Do-List do one or more of the following (i) cause changes to the y-structure to take into account the new position of the sweep line, (ii) add new ac-

tivities to the To-Do-List (iii) incrementally solve the complete problem. An example of the third type of activity is the checking and reporting of intersections. All plane sweep algorithms have the same basic control structure as given below in Figure 4.

```
Initialize the To-Do-List with basic information from the input;
y-structure ← ∅;
while To-Do-List is not empty do
    Do-It ← top
    case Do-It of
    {Each type of activity on the To-Do-List is listed.
     Each activity consists of one of more of the following:
        updating the y-structure,
        inserting new activities on the To-Do-List,
        producing output.}
    end
end.
```

Figure 4: Control structure of a plane sweep algorithm

There are several approaches to making a plane sweep algorithm efficient: (i) try to minimize the number of activities that are put on the To-Do-List, (ii) try to make the y-structure simple so that it is efficient to maintain, and (iii) make the y-structure comprehensive so that finding intersections with it is efficient. There is an obvious trade off between (ii) and (iii), making the y-structure simple makes maintaining it fast and using it to find intersections slow, while making the y-structure complex makes using it to find intersections fast and maintaining it slow. The major chord algorithm has a relatively simple y-structure and spends a large portion of its time checking intersections between objects. The chain algorithm, on the other hand, has a comprehensive y-structure and spends most of its time maintaining it. In any case an efficient data structure implementing the y-structure is desirable.

MV-trees

In our algorithms, the underlying data structures implementing the y-structure share a common feature. Both maintain families of subsets of an ordered set S and use the following operations.

- Maintenance
  - □ insert(x, V) – insert x into the set V,
  - □ delete(x, V) – delete x from the set V,
  - □ $W \leftarrow$ copy(V) – create a new set W which is identical to V.

- Queries
  - □ search(x, V) $\in \{T, F\}$ – if $x \in V$ then T else F,
  - □ list([$x..y$], V) $\subseteq S$ – the set of all the members of V which are between x and y inclusive,
  - □ count([$x..y$], V) $\in \mathbf{Z}^+$ – the number of members of V which are between x and y inclusive.

for $x, y \in S$ and $W, V \subseteq S$.

A standard search structure like a balanced binary search tree could be used to support all the operations except 'copy'. Several of the balanced binary tree algorithms can be modified to support copying efficiently as well as the standard operations. To this elegant modification we give the generic name *multiple version trees* (MV-trees). With MV-trees the 'insert' and 'delete' operations on a set of size n can be done in time O(log n) and 'copy' can be done in constant time. The 'search' and 'count' operations can also be done in O(log n) time. The 'list([$x..y$], V)' operation takes time O(log n + $|\{z \in V : x \leq z \leq y\}|$).

Consider a balanced binary tree algorithm that relies solely on balancing actions on the path of insertion or deletion. Dichromatic trees, 2-3 trees, and AVL-trees are examples of algorithms that can be modified to become multiple version trees [7, 6]. The task is done by judicious coping and sharing of nodes. A set is represented as a pointer to a balanced tree. The operation, $W \leftarrow$ copy(V), is implemented by simply setting the value of W to the value of V. Insertion and deletion are done in the usual way but new nodes must be allocated on the path from the root to the external node involved in the insertion or deletion. Each node that would have been modified by the algorithm is first copied then modified; the original is left intact. Figure 5 illustrates an example.

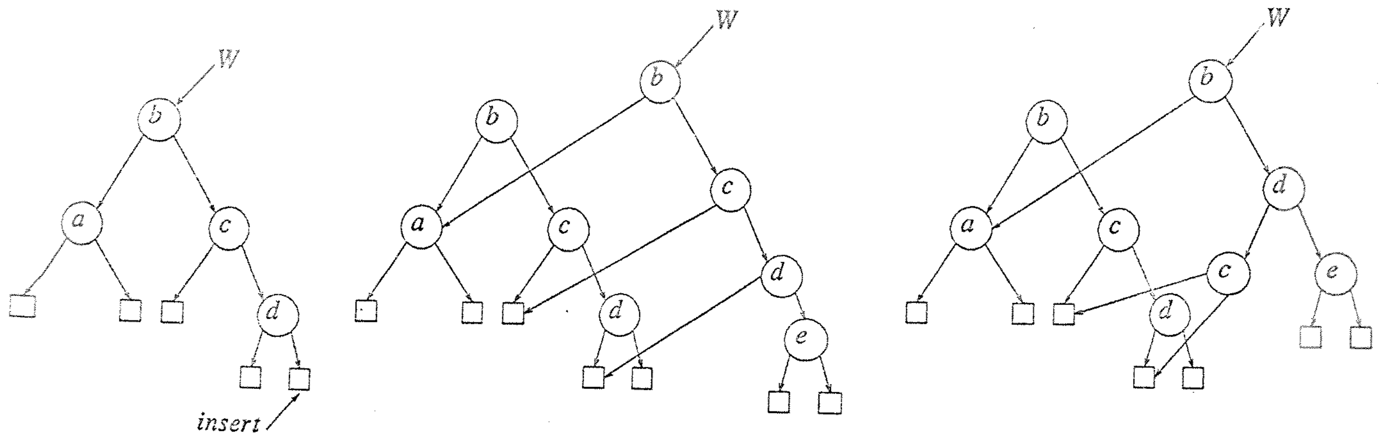


Figure 5: Insertion of a node with a rotation for balancing in an MV-tree

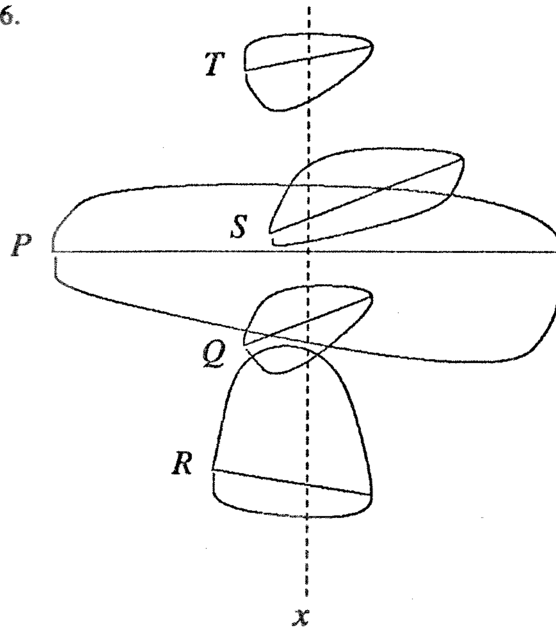
As the algorithm proceeds nodes can become unreachable. For example, in Figure 5 if nothing is

pointing to the root of the the tree after the insertion then the root has become unreachable. Because the data structure is inherently acyclic a simple reference counting scheme can reclaim all the unreachable nodes with no increase in the asymptotic running time.

3. Major Chord Algorithm

The major chord algorithm is based on a particular characterization of the pairwise intersections between convex objects. To explain this characterization we will use the following ideas in addition to that of the major chord.

Given a set of convex objects and a point x in the plane, the *objects at x* is an ordered set of objects which intersect the vertical line through x. This set is ordered by the y-coordinate of the intersection of the object's major chord and with the vertical line through x. The *interval of P at x* is the subset of the objects at x whose major chords intersect the object P. Note that this subset is an interval in the set of objects at x. The intervals of two objects are *adjacent* at x if the two intervals do not intersect but the union of the two intervals forms an interval in the set of objects at x. These concepts are illustrated in Figure 6.



The objects at x are (R, Q, P, S, T). The interval of P at x is (Q, P, S), the interval of R at x is just (R) and the intervals of P and R are adjacent at x.

Figure 6: Relationships between objects

Using these concepts we can characterize intersecting pairs of objects as follows.

LEMMA 1: If two convex objects P and Q intersect then there is an x such that the intervals of P and Q at x are adjacent or intersect.

PROOF: Assume to the contrary that P and Q intersect and there is no x such that the intervals of P and Q are adjacent or intersecting.

Pick any point p in the intersection of P and Q. As P is convex, a vertical line through p intersects P in a vertical line segment, similarly for Q. Since p is contained in both P and Q these vertical line segments themselves intersect. By assumption the intervals of P and Q are not adjacent or intersecting at any point and in particular at p there is at least one object R separating the interval of P and Q. Since the interval of P at p does not include R, P does not intersect the major chord of R at p. Likewise Q does not intersect the major chord of R at p. On a vertical line through p, P and Q are separated by the major chord of R, contradicting our assumption that P and Q intersect at p. []

We use this lemma as the basis of our algorithm. Instead of checking each pair of objects against each other for intersection we check only those objects whose intervals are adjacent or intersect. The y-structure maintains the information which allows us to check the conditions of the lemma easily. It maintains the ordered set of objects and the interval of each of the objects at x. The intervals are maintained in such a way that at any one time one interval may grow or shrink by one object. Thus for two intervals to intersect, they must start out intersecting or adjacent, or at one point they must *become* adjacent. In this way we need only check two objects P and Q for intersection when

- a new object P is being inserted in or adjacent to the interval of an object Q

or

- the intervals of P and Q become adjacent.

The y-structure can change only in the following situations:

1. Start of an object. At the left-most point of an object, it must be inserted into the set objects at the current sweep line position and inserted into the intervals of objects which contain the left-most point of the object.
2. Major chord-major chord intersection. When the major chords of two objects intersect, the order of the objects must be switched.
3. Boundary-major chord intersection. This occurs when an object Q intersects the major chord of P. This can happen in four different ways as illustrated in figure 7. In the first two ways the interval of Q expands, in the other two, the interval of Q contracts.
4. End of an object. When the right-most point of an object is reached, it must be removed from the set of objects.

An example of a set of objects, its activities and the evolution of its y-structure is given in figure 8.

Implementation of the To-Do-List

As the sweep progresses, the To-Do-List contains entries for each event which causes a change to the y-structure. The To-Do-List initially contains entries to indicate the start and end of each of the objects in the input. These are placed at the **x**-coordinate of the left and right ends of the major chord. The major chord of a convex object, denoted **mc**(P), may be found in time O(log k) using the standard technique of maximizing and minimizing a bimodal function (see Chazelle and Dobkin [3]). Once computed the major chord of each object is stored. Added to the To-Do-List during the plane

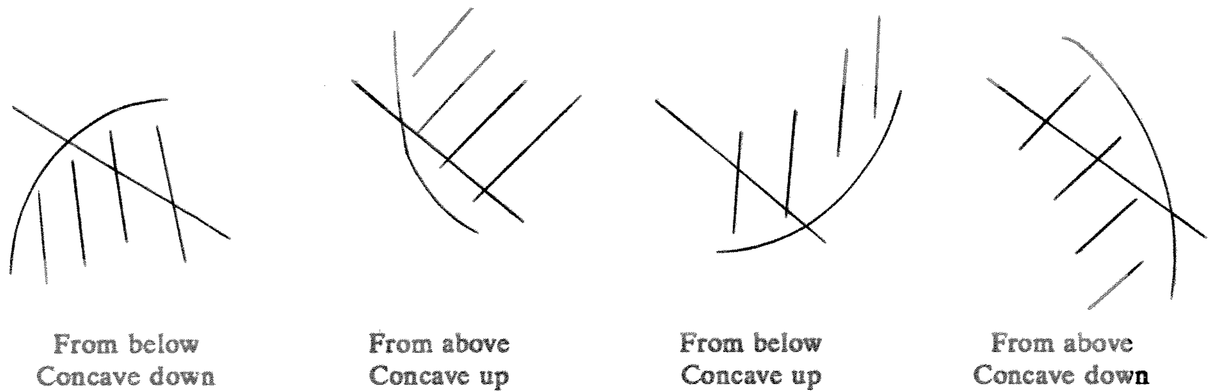


Figure 7: Four kinds of Boundary-major chord intersections.

sweep are entries for major chord-major chord and boundary-major chord intersections. These are recognized and placed on the To-Do-List when the objects become adjacent and when the object and the interval become adjacent respectively.

Implementation of the y-structure

The ordered set of objects at x is represented as a balanced tree. Also maintained are pointers to the objects currently above and below each object P, denoted $above(P)$ and $below(P)$ respectively.

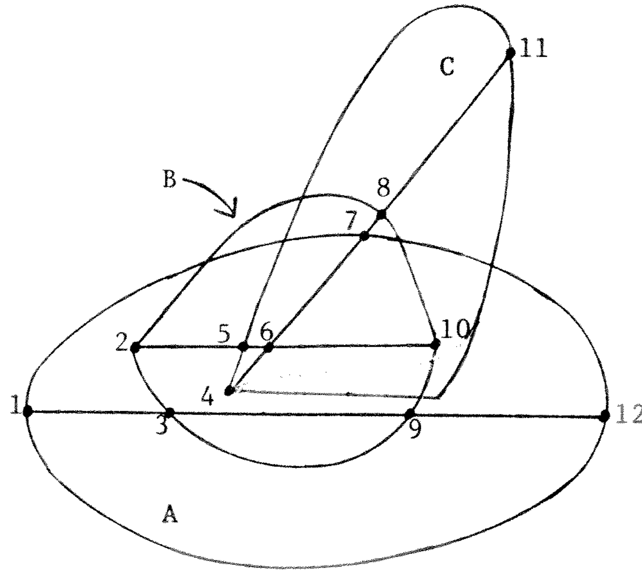
The intervals at x are maintained using three sets for every object at x. For each object P, $TOP(P)$ is the set of objects whose intervals have P as their largest element, $BOT(P)$ is the set of objects whose intervals have P as their smallest element and $MID(P)$ is the set of objects whose intervals contain P, but not as the largest or smallest element. The sets $TOP(P)$ and $BOT(P)$ are implemented as balanced trees, while $MID(P)$ is implemented as an MV-Tree. The ordering used in these sets is arbitrary and static (say based on their order within the input or in memory); this is distinct from the dynamic ordering of the set of objects at x.

Specification of actions

The algorithm will now be completely specified by giving the actions to be performed when each type of entry reaches the top of the To-Do-List. The total running time for each step is given in brackets at the end of the step. This does not include the time to insert entries onto the To-Do-List. The algorithm is analyzed in terms of the following quantities, some of which appeared in table 1.

n = number of objects
k = number of segments per object
s = number of object-object intersections
t = number of major chord-major chord intersections
u = number of boundary-major chord intersections
m = maximum cardinality of any individual TOP or BOT set

- Start of an Object P.
  1. Insert P into the ordered set of objects at x based on the y-coordinate of its left-most point. [$O(n \log n)$]



Activities and the y-structure
($\delta(A)$ is the boundary of A, mc(A) is the major chord of A)

i Activity at *i*	Ordered set of objects at *i*	Interval of *A* at *i*	Interval of *B* at *i*	Interval of *C* at *i*
1. Start of *A*	*A*	*A*		
2. Start of *B*	*A B*	*A B*	*B*	
3. Int. $\delta(B)$ and mc(A)	*A B*	*A B*	*A B*	
4. Start of *C*	*A C B*	*A C B*	*A B C*	*C*
5. Int. $\delta(C)$ and mc(B)	*A C B*	*A C B*	*A C B*	*C B*
6. Int. mc(B) and mc(C)	*A B C*	*A B C*	*A B C*	*B C*
7. Int. $\delta(A)$ and mc(C)	*A B C*	*A B*	*A B C*	*B C*
8. Int. $\delta(B)$ and mc(C)	*A B C*	*A B*	*A B*	*B C*
9. Int. $\delta(B)$ and mc(A)	*A B C*	*A B*	*B*	*B C*
10. End of *B*	*A C*	*A*		*C*
11. End of *C*	*A*	*A*		
12. End of *A*				

Figure 8: The steps of the Major Chord Algorithm

2. Check for intersection of **mc**(P) and **mc**($above(P)$) and insert a major chord-major chord intersection into the To-Do-List if they do. [$\Theta(n)$]

3. Similarly check **mc**(P) and **mc**($below(P)$). [$\Theta(n)$]

4. $MID(P) \leftarrow$ copy($MID(above(P))$). [$\Theta(n)$]

5. **for all** $Q \in TOP(above(P))$ **do** insert($Q, MID(P)$). [$O(n\ m\ log\ n)$]

6. Report intersection of P with all objects $Q \in MID(P)$. [$O(s)$]

7. $TOP(P) \leftarrow BOT(P) \leftarrow \{P\}$. [$\Theta(n)$]

8. Execute the following loop:

```
for all Q ∈ BOT(above(P)) do
    check P and Q for intersection and report it if they do;
    if Q and mc(P) intersect then
        insert a boundary-major chord intersection in the To-Do-List;
end
```

[$O(n\ m \log k)$]

- Major chord-major chord intersection between P and Q (P initially above Q).

1. Switch $BOT(Q)$ and $BOT(P)$, switch $TOP(Q)$ and $TOP(P)$ and switch P and Q in the ordering of objects at x. [$\Theta(t)$]

2. Check **mc**(P) against **mc**($below(P)$) inserting a major chord-major chord intersection in the To-Do-List if they do. [$\Theta(t)$]

3. Similarly check **mc**(Q) against **mc**($above(Q)$). [$\Theta(t)$]

- Boundary-major chord intersection between **mc**(P) and Q. Assume Q is coming from *below* and is concave *downwards*, the other three cases are similar.

1. delete(Q, $TOP(below(P))$); insert(Q, $TOP(P)$) and insert(Q, $MID(below(P))$). [$O(u \log n)$]

2. Check Q for intersection with **mc**($above(P)$) and insert a boundary-major chord intersection if they do. [$\Theta(u)$]

3. **for all** $R \in BOT(above(P))$ **do** check Q and R for intersection and report it if they do. [$O(u\ m \log k)$]

- End of object P

1. Execute the following loop:

```
for all Q ∈ BOT(above(P)) do
    for all R ∈ TOP(below(P)) do
        Check Q and R for intersection and report it if they do
    end
end.
```

[$O(n\ m^2 \log k)$]

2. **for all** $Q \in (TOP(P) - \{P\})$ **do** insert($Q, TOP(below(P))$). [$O(n\ m \log n)$]

3. **for all** $Q \in (BOT(P) - \{P\})$ **do** insert($Q, BOT(above(P))$). [$O(n\ m \log n)$]

4. Check for intersection of $below(P)$ with **mc**($above(P)$) and insert a boundary-major chord intersection into the To-Do-List if they do. [$\Theta(n)$]

5. Similarly check $above(P)$ with **mc**($below(P)$). [$\Theta(n)$]

Running time

The total running time of this algorithm including the time needed to initialize and maintain the To-Do-List is given by

$$O((t + u) \log n + n\, m^2 \log k + n\, m \log n + u\, m \log k + s).$$

This depends on the number of each type of intersection and the sizes of the intervals of the objects. No object may be in the intervals of any more than d objects, where d is the maximum depth of the overlap of objects, so we can bound m by d. The number of each type of intersection is bounded by the number of object-object intersections so t and u are bounded by s. We then have the following theorem.

THEOREM 2: The Intersection Reporting Problem for n convex polygons with $O(k)$ sides can be solved in time

$$O(n\, d^2 \log k + n\, d \log n + s\, d \log k),$$

where s is the number of intersections and d is the maximum depth of intersection.

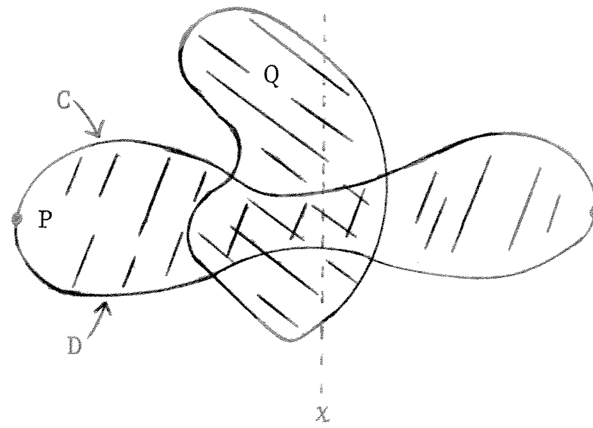
To eliminate the multiple reporting of object intersections an additional set for each object may be maintained to keep track of the objects which have been reported as intersecting. This data structure could be checked before reporting the intersection. Implementing this set as a balanced tree, the running time of the algorithm is bounded by

$$O((n\, d^2 + s\, d)(\log n + \log k)).$$

4. Chain Algorithm

This algorithm is based on a much simpler characterization of intersections between simple objects. Two objects intersect if and only if their boundaries intersect or one object is inside the other. We store the information needed to check these conditions in the y-structure. The y-structure of this algorithm indicates precisely which objects intersect any point on the sweep line. To implement this, let us define the following sets which will be maintained.

The *chains at x* is the ordered set of chains which intersect the vertical line through x. This set is ordered by the y-coordinate of the intersection of the chain with the vertical line through x. Given a chain C, the *family of C at x* is the set of objects which contain the half open segment of the vertical line through x starting from its intersection with C and going down to, but not including, its intersection with the chain below C in the ordering of the chains at x. An example of these ideas are given in figure 9.



The family of C is $\{P, Q\}$ and the family of D is $\{Q\}$.

Figure 9: Families of chains

Implementation of the To-Do-List

As the sweep progresses, the To-Do-List contains entries for each event which causes a change to the y-structure. The entries in the To-Do-List are initially: starts of chains; chain bends, a transition between one segment and the succeeding one within a chain; and ends of chains. These are inserted during an initial scan of the entire input. Added to the To-Do-List during the plane sweep are entries for segment-segment intersections. A segment-segment intersection is discovered when the chains the segments are a part of become adjacent in the set of chains (Bentley and Ottman [1] discuss reporting intersections of line segments). Since the boundary is topologically equivalent to a circle, chains start and end in pairs. There are two ways chains can start or end and these are shown below in figure 10 along with the other entries which may be placed on the To-Do-List. The input objects are split into chains and the chains are classified as part of the initial scan of the input.

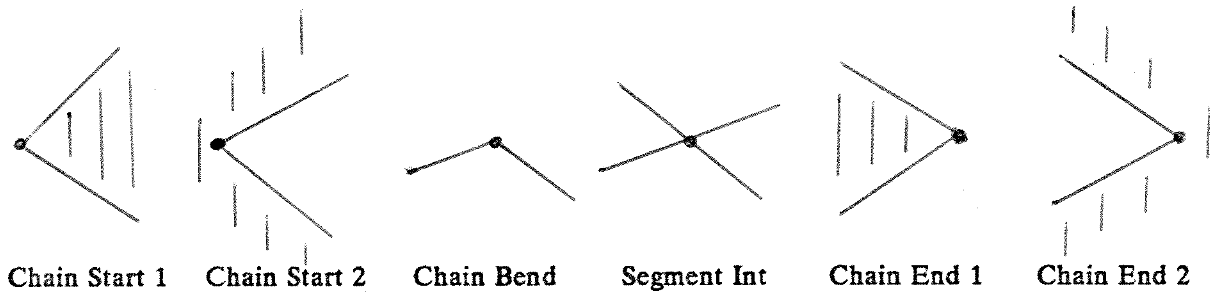


Figure 10: Entries on the To-Do-List in the Chain Algorithm

Implementation of the y-structure

The y-structure maintains the chains and their families at the current sweep line position. The ordered set of chains at x is represented as a balanced tree. Each chain has one segment which is active at any one time, the active segment is a segment intersected by the sweep line. For a chain C, we maintain a pointer to the current segment, denoted $seg(C)$, pointers to the chains above and below it in the ordering of chains at x, $above(C)$ and $below(C)$, and a pointer to the object the chain is a part of, $object(C)$. In addition, each chain can be categorized by whether the inside of the object is above or below it, if the latter then $top(C)$ is true, otherwise false. The family of a chain C is implemented as an MV-Tree, denoted as $FAM(C)$.

Specification of actions

The algorithm can be completely specified by the actions needed for every entry on the To-Do-List. This is given below along with the total running time for each step. The quantities used in the analysis, including those given in table 2 are

n = number of chains
k = number of segments per chain
p = total number of segments
s = number of object-object intersections (the things we are interested in!)
r = number of segment-segment intersections.

- Start of Chains type 1. (C is above D).
  1. Insert C and D into the y-structure based on their y-coordinate. [$O(n \log n)$]
  2. If this is first start of chains in this object then report intersection of $object(C)$ with everything in $FAM(above(C))$. $O(s)$
  3. $FAM(C) \leftarrow$ insert($object(C)$, copy($FAM(above(C))$)). [$O(n \log n)$]
  4. $FAM(D) \leftarrow$ copy($FAM(above(C))$). [$\Theta(n)$]
  5. Check for intersection between $seg(C)$ and $seg(above(C))$ and if so insert the segment-segment intersection into the To-Do-List. [$\Theta(n)$]
  6. Similarly check for intersection between $seg(D)$ and $seg(below(D))$. [$\Theta(n)$]
- Start of Chains 2. (C above D)
  1. Insert C and D into the y-structure based on their y-coordinate. [$O(n \log n)$]
  2. $FAM(C) \leftarrow$ delete($object(C)$, copy($FAM(above(C))$)). [$O(n \log n)$]
  3. $FAM(D) \leftarrow$ copy($FAM(above(C))$). [$\Theta(n)$]
  4. Check for intersection between $seg(C)$ and $seg(above(C))$ and if so insert the segment-segment intersection into the To-Do-List. [$\Theta(n)$]
  5. Similarly check for intersection between $seg(D)$ and $seg(below(D))$. [$\Theta(n)$]
- Bend in chain C, new segment is a.
  1. Update $seg(C) \leftarrow a$. [$\Theta(p)$]
  2. Check for intersection between $seg(C)$ and $seg(above(C))$ if so insert the segment-segment intersection into the To-Do-List. [$\Theta(p)$]
  3. Similarly check for intersection between $seg(C)$ and $seg(below(C))$. [$\Theta(p)$]

- Segment-segment intersection between C and D (C initially above D).
  1. Report intersection between *object*(C) and *object*(D). [$\Theta(r)$]
  2. Switch C and D in y-structure. [$\Theta(r)$]
  3. **if** *top*(C) **then** delete(*object*(C), *FAM*(D)) **else** insert(*object*(C), *FAM*(D)). [$O(r \log n)$]
  4. **if** *top*(D) **then** insert(*object*(D), *FAM*(C)) **else** delete(*object*(D), *FAM*(C)). [$O(r \log n)$]
  5. Check for intersection between *seg*(C) and *seg*(*below*(C)) and if so insert the segment-segment intersection into the To-Do-List. [$\Theta(r)$]
  6. Similarly check for intersection between *seg*(D) and *seg*(*above*(D)). [$\Theta(r)$]
- End of Chains 1 or 2.
  1. Remove C and D from the y-structure.

Running time

The total running time including initialization and maintenance of the To-Do-List is

$$O((p + n + r) \log p + (p + n) \log n + s).$$

Noting that $p = O(n\,k)$ we have the following theorem.

THEOREM 3: The Intersection Reporting Problem for n simple polygons with $O(k)$ sides can be solved in time

$$O((r + n\,k)(\log n + \log k) + s),$$

where s is the number of intersections and r is the number of segment-segment intersections.

To eliminate the multiple reporting of object intersections we can, as in the major chord algorithm, keep for each object the set of intersecting objects. The running time of the algorithm with this modification is asymptotically the same as above.

5. Conclusion

We have presented two plane sweep algorithms which solve much the same problem. The two algorithms have one general difference. The Major Chord Algorithm has relatively little information in its y-structure and spends a large portion of its time checking intersections between objects. The Chain Algorithm on the other hand has complete information in its y-structure and spends most of its time maintaining that structure. Further work may find a better compromise between the two, keeping enough information in the y-structure to quickly solve the problem while not incurring high maintenance costs.

The simple algorithms presented here illustrate some of the data structures and techniques which are extended to higher dimensions and to hidden line removal problems in a sequel.

References

[1] Bentley, J. L., Ottman, Th.
Algorithms for Reporting and Counting Geometric Intersections.
IEEE Transactions on Computers C-28:643-647, 1979.

[2] Bentley, J. L. and Wood, D.
An Optimal Worst Case Algorithm for Computing Intersections of Rectangles.
IEEE Transactions on Computers C-29:571-576, July, 1980.

[3] Chazelle, B. and Dobkin, D. P.
Detection is Easier Than Computation.
In *Proceedings of the 12th Annual Symposium on Theory of Computation.* ACM Special Interest Group on Automiton and Computability Theory, 1980.

[4] Dobkin, D. P. and Kirkpatrick, D. G.
Fast Detection of Polyhedral Intersections.
1982. Manuscript.

[5] Dobkin, D. P., Lipton, R. J.
On the Complexity of Computations Under Varying Sets of Primitives.
Journal of Computer and System Sciences 18:86-91, 1979.

[6] Guibas, L. J. and Sedgewick, R.
A Dichromatic Framework for Balanced Trees.
In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science.* IEEE Computer Society, 1978.

[7] Knuth, D. E.
The Art of Computer Programming. Volume 3: *Sorting and Searching.*
Addison-Wesley, Reading, Mass., 1973.

[8] McCreight, E. M.
Efficient Algorithms for Enumerating Intersecting Intervals and Rectangles.
Technical Report CSL-80-9, Xerox PARC, June, 1980.

[9] Nievergelt, J. and Preparata, F. P.
Plane-Sweep Algorithms for Intersecting Geometric Figures.
Communications of the ACM 25:739-747, 1982.

[10] Shamos, M. I. and Hoey, D.
Geometric Intersection Problems.
In *Proceedings of the 16th Annual Symposium on Foundations of Computer Science*, pages 151-162. IEEE Computer Society, 1975.

[11] Six, H.-W. and Wood, D.
The Rectangle Intersection Problem Revisited.
BIT 20:426-433, 1980.

[12] Vaishnavi, V. K. and Wood, D.
Rectilinear Line Segment Intersection, Layered Segment Trees and Dynamization.
Journal of Algorithms 3:160-176, 1982.