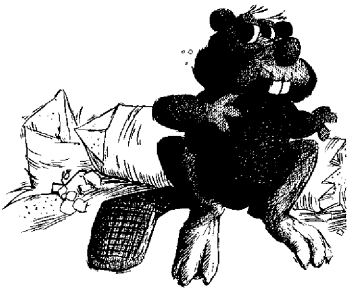
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



Computing the Connected Components of Simple Rectilinear Geometrical Objects In D-Space

Herbert Edelsbrunner
Jan van Leeuwen
Thomas Ottmann
Derick Wood

Data Structuring Group
CS-83-17

June, 1983

COMPUTING THE CONNECTED COMPONENTS OF SIMPLE RECTILINEAR GEOMETRICAL OBJECTS IN D-SPACE[1]

Herbert Edelsbrunner[2]

Jan van Leeuwen[3]

Thomas Ottmann[4]

Derick Wood[5]

ABSTRACT

Two or more geometrical objects (solids) are said to be connected whenever their union is a connected point set in the usual sense. Sets of geometrical objects are naturally divided into connected components, which are maximal connected subsets. We show that the connected components of a given collection of n horizontal and vertical line segments in the plane can be computed in $O(n \log n)$ time and $O(n)$ space and prove that this is essentially optimal. The result is generalized to compute the connected components of a set of n rectilinearly-oriented rectangles in the plane with the same time and space bounds. Several extensions of the results to higher dimensions and to dynamic sets of objects are discussed.

---

[1] This work was partially supported by a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692 and was carried out while the authors attended "Effiziente Algorithmen und Datenstrukturen" Oberwolfach, February 1981.

[2] Institutes für Informationsverarbeitung, Technical University Graz, Schieβstattgasse 4A, A-8010 Graz, Austria.

[3] Vakgroep informatica, University of Utrecht, Princetonplein 5, Postbus 80.002, 3508 TA Utrecht, The Netherlands.

[4] Universität Karlsruhe, Institut für Angewandte Informatik und Formale Beschreibungsverfahren, Universität Karlsruhe, Postfach 6380, D-7500, Karlsruhe, W. Germany.

[5] Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada.

1. INTRODUCTION

We say that two planar geometrical objects *intersect* or overlap if they have at least one point in common. Given a set of such objects its *connected components* are defined as the equivalence classes of the relation defined as the reflexive transitive closure of the intersection relation. Moreover given such a set the following five connectedness problems arise naturally:

(a) Do the objects form a single connected component?
(b) Are the objects completely disconnected?
(c) How many connected components are there?
(d) Which is the largest component?
(e) Which is the largest pairwise connected component?
(f) What are the connected components?

It should be clear that a solution for problem (f) also solves problems (a)-(d), although not necessarily in an optimal fashion. Problem (e) can also be solved beginning with the solution for problem (f) and checking which components include all pairwise connected objects. Hence we restrict out attention to problem (f), which we call the *Connected Components Problem.*

In the present paper, we primarily consider rectilinearly-oriented line segments and rectangles in the real plane (2-space, for short), that is only consisting of line segments parallel to one or other of the coordinate axes. Therefore, from here on in, we assume all line segments and rectangles under discussion to be rectilinearly-oriented.

Extending our techniques to arbitrary planar geometrical objects can be done as in Edelsbrunner [E2], that is using bounding boxes. In particular the approach presented in Section 3 for rectangles can be adapted, using the bounding box approach, to deal with arbitrary objects. We conjecture that this method will perform well in most cases.

If one can easily decide whether or not two objects of a given type are connected (as in the case of line segments and rectangles), then one can consider computing the transitive closure of the connectedness relation by standard methods, for example see [AHU]. However since there are $O(n^2)$ pairwise intersections, such an algorithm would require $\Omega(n^2)$ time in the worst case. Thus we are led to iteratively computing connected components using the sweeping line technique, a powerful paradigm, first used by [SH] in computational geometry, and subsequently considered in its own right in [NP]. This we detail in Section 2, while in Section 3 we show how the algorithm can be generalized to solve the same problem for rectangles. In both cases $O(n \log n)$ time and $O(n)$ space are achieved. The algorithm will be shown to be essentially optimal. In Section 4 we discuss how the connected component problem can be solved for d-ranges (d-dimensional rectangles) in d-space for $d \geq 3$. In this case a completely different approach seems to be appropriate.

It should be pointed out that we only consider static sets of objects in this paper. To deal with dynamic sets, some of the techniques of [OL] may be applied. None of the results seem to be anywhere near optimal and we leave the dynamic case open for further investigation.

Finally, in [IA] a solution to problem (e) for rectangles has been derived. Moreover they point out that problem (e) is the maximal clique problem for the corresponding intersection graph.

2. HORIZONTAL AND VERTICAL LINE SEGMENTS

Let an arbitrary set of n horizontal and vertical line-segments be given. We shall derive an $O(n \log n)$ time and $O(n)$ space algorithm to compute its connected components. Clearly the space requirement is optimal. But so is the time bound, as the following observation shows:

Observation 2.1: Computation of the connected components of a set of n horizontal and vertical line segments requires $\Omega(n \log n)$ time in the worst case.

Proof: Consider the very special case in which the line segments have length 0. Any algorithm to compute the connected components in this case would solve the ELEMENT UNIQUENESS problem for the set of points, which is known to require $\Omega(n \log n)$ steps in the worst case in a very reasonable model of computation (see [DL]). □

The algorithm we shall present computes the connected components using the plane sweep paradigm that has been successfully used for a number of similar problems (consult Nievergelt and Preparata [NP] for an excellent discussion of this paradigm.)

While the sweeping line scans the plane from left to right, a data structure is maintained in which the active components of the line segments met up until now are represented. A *component* is *active* if it intersects the sweeping line, that is if it contains at least one horizontal segment which intersects the sweeping line. Three operations need to be supported and these are

(i) the insertion of a component,

(ii) the deletion of a component, and

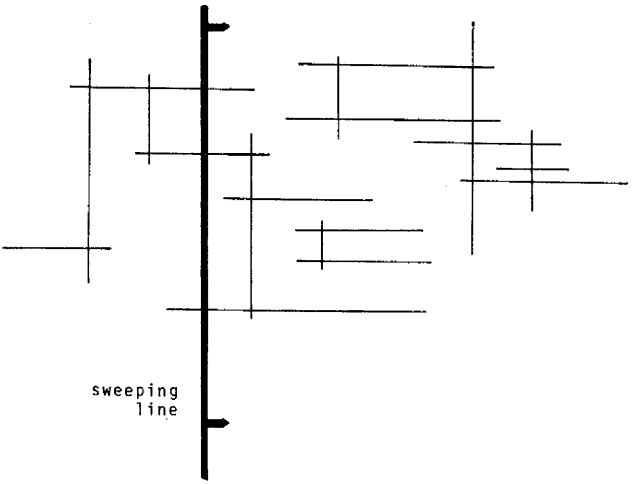
(iii) the detection of all active components that intersect a vertical line segment and the merging of these components.

Let us first present the overall structure of the algorithm and then discuss the accompanying data structure and the three operations in more detail.

A line segment will always be represented as $[x_1, y_1, x_2, y_2]$, where (x_1, y_1) and (x_2, y_2) are its endpoints and $(x_1, y_1) \leq (x_2, y_2)$ in the usual lexicographical ordering of the plane. We will assume for the sake of simplicity that for every two given vertical line segments $[x_A, y_A^{(1)}, x_A, y_A^{(2)}]$ $(y_A^{(1)} \leq y_A^{(2)})$ and $[x_B, y_B^{(1)}, x_B, y_B^{(2)}]$ $(y_B^{(1)} \leq y_B^{(2)})$ we have $x_A \neq x_B$ and that, likewise, the y-coordinates corresponding to different horizontal line segments are different. The restrictions will be removed at the end of this section. Note that we do not make a distinction between points, that is line

segments of length 0, and non-degenerate line segments.

The algorithm begins by sorting the x-coordinates of all line segments in the set. The sweeping line proceeds down the sorted list of values, the sweeping points, and carries out an operation, depending on whether the coordinate value represents the left or right end of a horizontal line segment or the position of a vertical line segment. See Figure 2.1.



Plane-sweep from left to right

Figure 2.1

Whenever we only represent a single coordinate value, as above and throughout the paper, we will always augment the representation so that we can retrieve the entire line segment to which it belongs in O(1) time.

At each position of the sweeping line the line segments which it has met so far determine connected sub-components of the components of the set. These are of two kinds. First, there are those sub-components which are, indeed, connected components of the set and are wholly to the left of the sweeping line. These are said to be *inactive* components, since they cannot be further changed. The other sub-components are said to be *active*, since at least one line segment in each of them cuts the sweeping line and, moreover, they are, potentially, incomplete.

The following observation summarizes the essential idea of the plane sweep algorithm.

Observation 2.2: Two disjoint components have only to be *merged* when a vertical line segment which intersects both of them is met.

The plane sweep algorithm carries out one of the following three operations on processing a new x-coordinate.

(1) If it is the left end of a horizontal line segment, this line segment is inserted into the supporting data structure as a new connected component.

(2) If it is the right end of a horizontal line segment, this line segment is deleted from the supporting data structure. However, since this line segment is a member of exactly one component only this component needs to be changed. This might lead to an active connected component becoming inactive, but in general this is not the case.

(3) If it is the x-coordinate of a vertical line segment a query is invoked asking for all active connected components which intersect it. The determined components together with the vertical line segment are *merged* into one new connected component.

The algorithm takes the obvious measures to maintain the validity of the set of components it keeps, when a next "endpoint", etc. is reached. Thus its correctness is assured as long as the supporting data structure and its operations are specified correctly.

The following paragraphs are devoted to a description of the data structure which supports the three operations of insertion, deletion and query defined above.

The data structure consists of three basic parts corresponding to three different representations of the components.

(I) The connected components, as they are known up to a given point, will be represented as disjoint sets as in any solution to the common UNION-FIND problem (cf. [AHU]). We call this the *union-find* structure. We never remove components from this structure, hence on termination it contains all the connected components. Since there can be at most n connected components when given n line segments, there can be at most $n-1$ UNION operations. Also the algorithm will never execute a FIND operation, since we shall determine the component an object belongs to by using the other structures. Hence, using one of the structures proposed in [AHU] allows this to be implemented in $O(n)$ time and space.

(II) Each active connected component is represented by the interval defined by its current topmost and bottommost intersection with the sweeping line, which demarcate the *active interval* of the component. This active interval is, in fact, determined by active horizontal line segments in the connected component. These active intervals are represented in two distinct dynamic data structures, which together are called the *active structure*. The first is a balanced search tree which houses the y-coordinates of the endpoints of the active intervals, for example an AVL tree, see [AHU]. The structure not only cross-references the endpoints of the active intervals, but also their components. We call this structure the *intersection tree*. It supports insertions and deletions of active intervals in $O(\log n)$ time. Moreover given a y-interval, all active intervals, currently in the structure, which it intersects or encloses, but is not enclosed by, can be determined in $O(\log n+k)$ time, where k is the number of reported active intervals. The second structure is again a balanced search tree which houses the y-coordinates of the endpoints of the active intervals. However, in this case the nesting structure of the active intervals is represented. That the active intervals do, in fact, have a nesting structure is the substance of the following:

Lemma 2.3: Let S denote the set of active intervals on the sweeping line at some sweeping point, and let i and j denote two arbitrary active intervals in S.

Then either i and j are disjoint or one encloses to other.

Proof: Any two points x and y on the sweeping line that belong to the same component are connected by a simple plane curve tracing segments of the current component only. Consider two active intervals i and j on S, and suppose that they are neither disjoint nor nested, then there are points α_i and β_i on i and α_j and β_j on j such that, when using the linear ordering of points on S

$$\alpha_i \notin [\alpha_j, \beta_j], \quad \beta_j \notin [\alpha_i, \beta_i], \quad \text{and} \quad [\alpha_i, \beta_i] \cap [\alpha_j, \beta_j] \neq \emptyset .$$

It follows that the connecting curves of α_i and β_i and α_j and β_j must intersect, contradicting the fact that their components are disjoint. □

This lemma leads to the following:

Observation 2.4 All active intervals which enclose the query

interval are totally ordered with respect to enclosure. Consequently, only the smallest enclosing interval may potentially be associated with a component which intersects the vertical line segment corresponding to the query interval.

The purpose of the second structure, which we assume to be implemented by the *parenthesis tree* [GW], is to find the smallest active interval enclosing a given query y-interval. This query requires $O(\log n)$ time as does insertion and deletion. This could also be implemented by way of priority search tree [M].

(III) For each active component we maintain the left endpoints (called *ghost points*) of those horizontal line segments in it which still intersect the sweeping line and (thus) extend to its right. Effectively, the y-coordinates of the ghost points immediately tell us at what points the given component currently intersects the sweeping line. We call this structure the *ghost structure*. In Figure 2.2 A and D define an active interval while B and C are the ghost points of the active connected component.

The ghost structure consists of a balanced search tree, *the ghost tree*, for each active connected component. Each such tree contains the ghost points of the component, represented by their y-coordinates. Clearly such a structure allows an insertion and a deletion to be carried out in $O(\log n)$ time. Given a query y-interval and an active connected component, the query determines whether or not one of the ghost points of the component lies within the y-interval. This requires $O(\log n)$ time.

Having defined the various supporting data structures, we now are in a position to consider the three possible actions of the plane sweep algorithm in more detail.

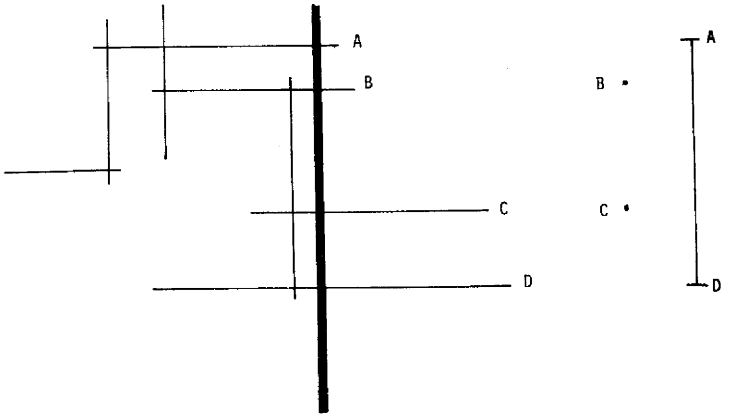
2.1. On Meeting a Horizontal Line Segment

Horizontal line segments serve to introduce new active components and to possibly transform an active component into an inactive one.

On meeting the left endpoint of a horizontal line segment add a new connected component to the union-find structure and add a new active interval to the active structure. No ghost points are created at this time.

On meeting the right endpoint of a horizontal line segment, the point must be deleted from either the active structure of from the ghost structure.

In the former case one of three possibilities occurs:



Connected component with its active interval and ghost points

Figure 2.2

(i) the active interval is determined solely by this line segment, in which case its component becomes inactive,

(ii) the active interval collapses into a single point since the ghost structure for this component is empty, or

(iii) the ghost structure for this component is not empty, in which case the closest ghost point becomes the new endpoint of the active interval.

Observe that the closest ghost point is the ghost point with either minimum or maximum y-coordinate in the ghost tree for this component. Hence it can be determined and deleted from the ghost tree in $O(\log n)$ time.

2.2. On Meeting a Vertical Line Segment

Disjoint active components are merged whenever a vertical line segment (that is a *query interval*) is met which is connected to both of them. To see how these disjoint components are to be determined on meeting such a query interval we make use of Lemma 2.3, Observation 2.4, and the following:

Observation 2.5 If an active interval intersects the query interval but does not enclose it then the associated component intersects the vertical line segment corresponding to the query interval.

This suggests the following strategy to determine all components which intersect a query interval.

First, determine all active intervals which intersect the query interval, but do not enclose it. For this purpose use the intersection tree.

Second, determine the smallest active interval enclosing the query interval using the parenthesis tree. If such an interval exists, query its corresponding ghost tree to determine whether or not any of its ghost points lie within the query interval.

By our previous comments these queries require $O(\log n+k)$ time in the worst case, where k is the number of reported active components.

If no active component is found, then the vertical segment corresponding to the query interval is a connected component in its own right (no future line segment can intersect it because of our insistence on unique x- and y-coordinate values). Hence it should be added to the union-find structure as an inactive singleton component.

If only one intersecting active component is found, then no action need be taken, apart from adding the line segment to the active component in the union-find structure.

In the remaining case at least two active components, which intersect the query interval, have been found. Thus the corresponding disjoint sets of line segments in the union-find structure need to be merged, and the vertical line segment is added to the resulting set. However this is not all. The active and ghost structures need to be updated also. We consider the merger of two components, since the merger of more than two can be always be realized as a sequence of these simpler mergers.

In the active structure, the two inner endpoints of the two active intervals must be deleted and re-inserted as ghost points in the ghost structure after it has been modified. Also the representation of the two outer endpoints must be modified to reflect their new rôle. Each update requires $O(\log n)$ time.

In the ghost structure the two sets of ghost points also should be merged to give one new ghost tree. Fortunately because of the observed properties of active intervals we find that the only kinds of mergers that can occur are:

(1) The merger of two disjoint sets in which all the points in one precede all the points in the other.

(2) The merger of two disjoint sets in which the points in one fall in between two points in the other.

Alternative (2) corresponds to a SPLIT followed by two MERGE operations.

A structure which supports these five operations is known as a concatenable queue and in [AHU] it is proved that height-balanced trees support each of these operations in $O(\log n)$ time using $O(n)$ space.

2.3. Removing the Restrictions

We are only left with the problem of how to relax the restrictions on endpoint x- and y-coordinate values. Two horizontal (vertical) line segments which have the same y- (x-) coordinate may cause troubles only if they overlap. But these line segments can be merged into a new line segment in a preprocessing step of the algorithm. It is readily seen that $O(n \log n)$ time suffices to merge all the troublesome line segments.

To summarize the results of this section we have demonstrated:

Theorem 2.6 *The connected components problem for n vertical and horizontal line segments in the plane can be solved in* $O(n\log n)$ *time and* $O(n)$ *space.*

3. RECTANGLES

We assume the rectangles are presented as a quadruple defining their four corner points. In following through the plane-sweep approach we see that only two cases (as against three for line segments) have to be treated, namely:

(i) meeting the left end of a rectangle, and

(ii) meeting the right end of a rectangle.

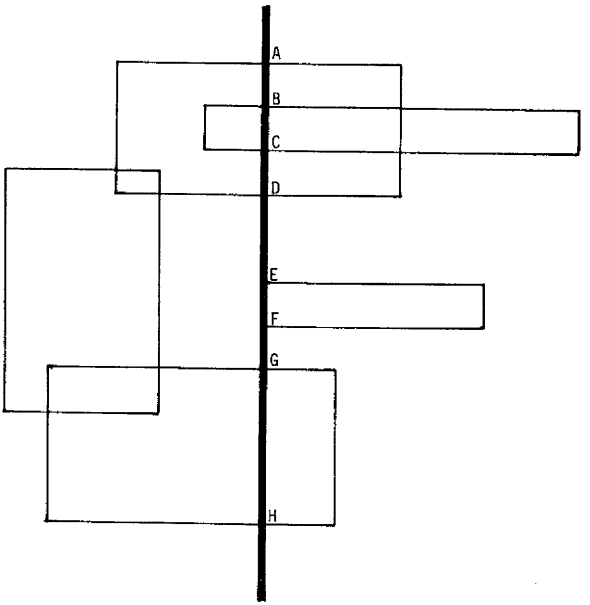
We once more assume that we have three structures, a union-find, an active and a ghost structure to represent the components met so far. Although the union-find and active structure are identical the ghost structure does have differences since it must deal with ghost intervals rather than points. For example in Figure 3.1, A-H defines one active interval, while A-D, B-C and G-H define its ghost intervals. Fortunately Observations 2.4 and 2.5 still hold and hence the strategy outlined for vertical line segments in Section 2.2 still holds. Thus the modified ghost tree has to be able to support the query:

Is there a ghost interval which intersects the given query interval?

Observe that this query can be solved in O(1) time, when given the result of the following query:

How many ghost intervals intersect the given query interval?

This is called a stabbing query, and in [GMW] a balanced tree structure is provided which enables such queries to be answered in $O(\log n)$ time and also allows an insertion or a deletion to be carried out in $O(\log n)$ time.



Active and ghost intervals of rectangles

Figure 3.1

If no ghost interval intersects the query interval then it forms (together with the connected components detected by the active structure) the basis of a new component and the appropriate structures need to be updated. Otherwise it forms an additional ghost interval of an active component which consists of the detected components.

On meeting the right end of a rectangle, the corresponding ghost or active interval is deleted. In the latter case a new active interval is obtained, possibly empty.

Hence we have shown:

Theorem 3.1 *The connected components problem for* n *rectilinear rectangles can be solved in* $O(n \log n)$ *time and* $O(n)$ *space.*

4. GENERALIZATION TO HIGHER DIMENSIONS

The obvious generalization of the algorithms presented in the previous sections is to sweep the d-dimensional space, for d no less than three, by a $(d-1)$-dimensional hyperplane. However, we do not know how to implement these algorithms efficiently. Because of this we use a simpler approach which we now sketch.

Initially insert all n d-dimensional rectilinear objects in some data structure D, that efficiently supports intersection queries and deletions of objects. Such structures have been described by Six and Wood [SW], Edelsbrunner [E1], and Edelsbrunner and Maurer [EM] and a method to speed-up deletions was developed by Overmars and van Leeuwen [OL]. Additionally, choose an arbitrary object, let it be the only member of a queue and delete it from D. (This object is initially called the *next* object in the queue.)

A single step takes the next object in the queue, performs an intersection query with it on D, inserts all intersecting objects into the queue, and deletes them from D.

This single step is performed until the queue is empty. At this stage all objects that have been added to the queue form one connected component. If D is now empty the algorithm terminates, otherwise the whole process is repeated to find the next connected component.

From the results presented in Edelsbrunner [E1] and Edelsbrunner and Maurer [EM] we know that the above algorithm can be carried out in $O(n\log^d n)$ time and requiring $O(n\log^{d-1} n)$ space for n d-dimensional rectilinear rectangles. However, whether or not this is optimal remains as an open problem.

5. REFERENCES

[AHU] Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co., Inc., Reading, Mass., 1974.

[DL] Dobkin, D., and Lipton, R., On the Complexity of Computations under Varying Sets of Primitives, *Journal of Computer and System Sciences 18* (1979), 86-91.

[E1] Edelsbrunner, H., Dynamic Data Structures for Orthogonal Intersection Queries, Technical University Graz, Institut für Informationsverarbeitung Report 59, T980.

[E2] Edelsbrunner, H., Reporting Intersections of Geometric Objects by Means of Covering Rectangles, *Bulletin of the EATCS*, (1980),

[EM] Edelsbrunner, H., and Maurer, H.A., On the Intersection of Orthogonal Objects, *Information Processing Letters 13* (1981), 177-181.

[GMW] Gonnet, G.H., Munro, J.I., and Wood, D., Direct Dynamic Data Structures for Some Line Segment Problems, *Computer Graphics and Image Processing*, (1983), to appear.

[GW] Güting, R.H., and Wood, D., The Parenthesis Tree, *Information Sciences 27* (1982), 151-162.

[IA] Imai, H., and Asano, T. Finding the Connected Components and a Maximum Clique of an Intersection Graph of Rectangles in the Plane, Technical Report, University of Tokyo, 1981.

[M] McCreight, E.M., Priority Search Trees, Xerox Palo Alto Research Centers Report CSL-81-5, 1982.

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

[OL] Overmars, M.H., and van Leeuwen, J., Worst Case Optimal Insertion and Deletion Methods for Decomposable Searching Problems, *Information Processing Letters 12* (1981), 168-173.

[SH] Shamos, M.I., and Hoey, D., Geometric Intersection Problems, *Proceedings of the 17th Annual IEEE FOCS Symposium*, (1976), 208-215.

[SW] Six, H.-W., and Wood, D., Counting and Reporting Intersections of d-Ranges, *IEEE Transactions on Computers C-31*, (1982), 181-187.