OPTIMAL ALGORITHMS FOR COMPUTING CONNECTED COMPONENTS OF BICHROMATIC LINE SEGMENTS AND POLYGONS

David Dobkin
Jenny Zehong Zhao

CS-TR-366-92

(March 1992)

Optimal Algorithms for Computing Connected Components

of Bichromatic Line Segments and Polygons [1]

David Dobkin and Jenny Zehong Zhao
Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract

A set of planar geometrical objects can be partitioned into connected components, where these are defined by the reflexive transitive closure of the pairwise intersection or overlap relation. We consider the problem of finding connected components of the union between m blue line segments and n red line segments in the plane, assuming that two line segments are disjoint if they have the same color. We solve this problem in $O(N \log N)$ time and $O(N)$ space, where $N = m+n$, by using a variant of the segment tree and union-find technique. It is then generalized to determine, in the same time and space bounds, the connected components of bichromatic simple polygons with N sides total, where polygons with the same color do not intersect or overlap.

Keywords. Computational geometry, line segment, connected components, interval graphs.

[1]This work was supported in part by the National Science Foundation under Grant Number CCR90-02352

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. The connected component problem is to find the connected components for a given set of objects.

In this paper, we will study the connected component problem for N bichromatic line segments. A set of line segments is called bichromatic if they can be partitioned into two sets such that line segments in the same set do not intersect with each other. Edelsbrunner, van Leeuwen, Ottmann and Wood solved connected component problem for n orthogonal line segments in $O(n \log n)$ time and linear space by line sweeping [ELOW]. Our main result is an algorithm that solves the connected component problem for bichromatic line segments in $O(N \log N)$ time and $O(N)$ space. It improves a previous $O(N \log^2 N)$ result by Guibas, Overmars, and Sharir. They also presented an $O(n^{4/3+e})$ time algorithm that finds connected components for n line segments in general position [GOS]. Later on we extended our first algorithm to solve the problem for bichromatic polygons.

Obviously we can first compute the intersections of a set of n line segments and then form its reflexive transitive closure by standard techniques. However since there may be $\Omega(n^2)$ pairwise intersections, such an algorithm would require time $\Omega(n^2)$. We will show how to fine the connected components without explicitly constructing all the edges. The data structure we use is a modification of a segment tree. We generate the result without explicitly going through every intersection. The detail is shown in section 2, in section 3 we show how the algorithm can be generalized to solve the same problem for bichromatic polygons. The same time and space bound is achieved, where N is the total number of sides of the polygons. It is easy to see that our algorithm can solve the connected component problem for N orthogonal line segments in the same time and space bound as in [ELOW] by coloring the vertical line segments blue and horizontal line segments red. The time and space optimality follows from the fact that the connected component problem for N orthogonal line segments has a lower bound of $\Omega(N \log N)$.

2 Bichromatic Line Segments

2.1 Data Structure

In this section, we introduce our basic data structure, the segment tree. Recall in the problem we are about to solve, we are given two sets of line segments, $B = \{b_1, b_2, ..., b_m\}$ and $R = \{r_1, r_2, ..., r_n\}$, of m blue and n red line segments in the plane. Any two line segments of the same color are disjoint. Our goal is to compute the connected components of the $m + n$ line segments without having to go through all line intersection points individually. Define $N = m + n$. We will use the segment tree described in [CEGS] and the union-find operation to solve the problem in $O(N \log^2 N)$ time and $O(N \log N)$ space. Later on we will show how the fractional cascading and topological sorting technique can be used to improve the time bound to $O(N \log N)$ with a linear space cost.

Our first step is to construct a segment tree, $\mathcal{T}$, on the interval decomposition of the x-axis induced by the x-coordinates of the endpoints of the given line segments. More specifically, $\mathcal{T}$ is

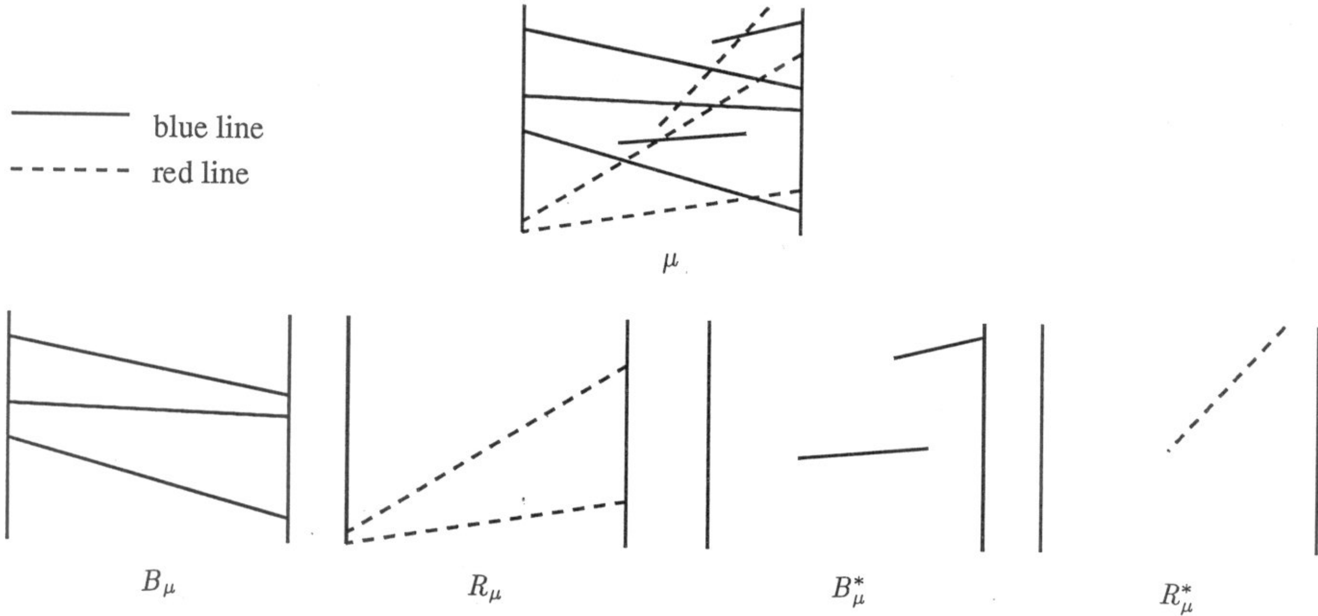

Figure 2.1:

defined as follows. Let us assume that the x-coordinates of the $2N$ end points are pairwise disjoint. If they are not we can get this property by simulating an arbitrarily small perturbation of the x-coordinates. As a result, the $2N$ x-coordinates decompose the x-axis into $2N + 1$ (atomic) intervals. $\mathcal{T}$ is a minimum height ordered binary tree whose ith leaf corresponds to the ith atomic interval from the left. Each interior node, μ, represents an interval, I_μ, that is the union of the intervals associated with the leaves of the subtree rooted at μ. Alternatively, we can think of μ as representing the vertical slab $\sigma_\mu = \{(x,y)|x \in I_\mu, y \in \mathcal{R}\}$. In the standard segment tree, each node μ (internal and external) has an associated list, L_μ, of all line segments, s, with the property that the vertical projection of s contains I_μ but does not contain I_k, with k the parent of μ. In our version, we maintain four lists with each node μ. First we split L_μ into two *standard lists*, B_μ and R_μ, of the blue and red line segments in L_μ. Furthermore, we associate with μ two additional so-called *hereditary lists*, B^*_μ and R^*_μ. B^*_μ contains all blue line segments stored in L_ν for a proper descendant ν of μ, and R^*_μ contains all such red line segments. In other words, whenever we store a line segment s in some B_ν or R_ν, we also store the line segment in the hereditary list of each proper ancestor μ of ν (see Figure 2.1).

For the applications that follow, it is helpful to regard each element of a standard or hereditary list of a node μ as representing the subsegment $s \cap \sigma_\mu$ of the corresponding line segment s. As suggested by this interpretation, we refer to line segments stored in the standard lists as *long segments* and to those stored in the hereditary lists as *short segments.*

In a standard segment tree, the total size of all the lists L_μ is $O(N \log N)$. Our first observation is that this is also the total size of all hereditary lists. Indeed, each line segment s is stored in $O(\log N)$ standard lists. The interior nodes that store s in their hereditary lists form two paths both starting at the root of $\mathcal{T}$; thus s is stored in at most $O(\log N)$ hereditary lists.

2.2 Algorithm

Next we show how to use the segment tree to solve the connected component problem for bichromatic line segments.

The input to the algorithm is a set of red line segments R and a set of blue line segments B where the same colored line segments don't intersect each other. The output of the algorithm is a list of connected components each of which is represented by line segments within. The outline of the algorithm is the following:

Algorithm.
1. Construct the standard tree structure for UNION-FIND operation
by creating a single tree for each line segment in B and R.
2. Construct the segment tree $\mathcal{T}$ for B and R.
3. **For** each node μ in $\mathcal{T}$ **do**
begin
3.1 sort B_μ and R_μ in decreasing y coordinates.
3.2 merge sort the two lists twice. once according to the left end points,
once according to the right end points, respectively
3.3 construct an interval graph for all blue segments in B_μ
from the resulting sorted list.
3.4 sweep the interval graph to form the connected components.
3.5 sort B^*_μ and R_μ by binary searching the position
in R_μ for left end point and right end point of each
line segment in B^*_μ.
3.6 repeat 3.3 and 3.4 for the new sorted list.
3.7 repeat 3.5 and 3.6 for R^*_μ and B_μ.
End.

4. Traverse the resulting trees from UNION-FIND operation to report the connected components. Each tree represents one connected component.

End of Algorithm.

Now we will explain in detail some of the steps in the algorithm.

3.3 Let the sorted list $B_\mu = (b_1, b_2, ..., b_k)$, and $R_\mu = (r_1, r_2, ..., r_l)$. For each blue segment b, suppose its left end point is between r_i and r_{i+1} and its right end point is between r_j and r_{j+1}, If $i = j$, then b does not intersect any red line segment in μ. Otherwise b intersects with $r_{min(i,j)+1}$, ..., $r_{max(i,j)}$, and we associate b with interval $[min(i,j)+1, max(i,j)]$. We create a new coordinate system. The horizontal axis represents the index of R_μ and the vertical axis represents the index of B_μ. Then we plot each blue segment with a horizontal line segment that spans the corresponding interval. The resulting graph is called interval graph (see Figure 2.2).

3.4 Starting from the minimum bound of horizontal range in the interval graph, we sweep to the right. The event list consists of line segments that intersect the sweeping line currently. If we encounter a left end point of blue segment, we add it to the event list. Do a UNION-FIND on this

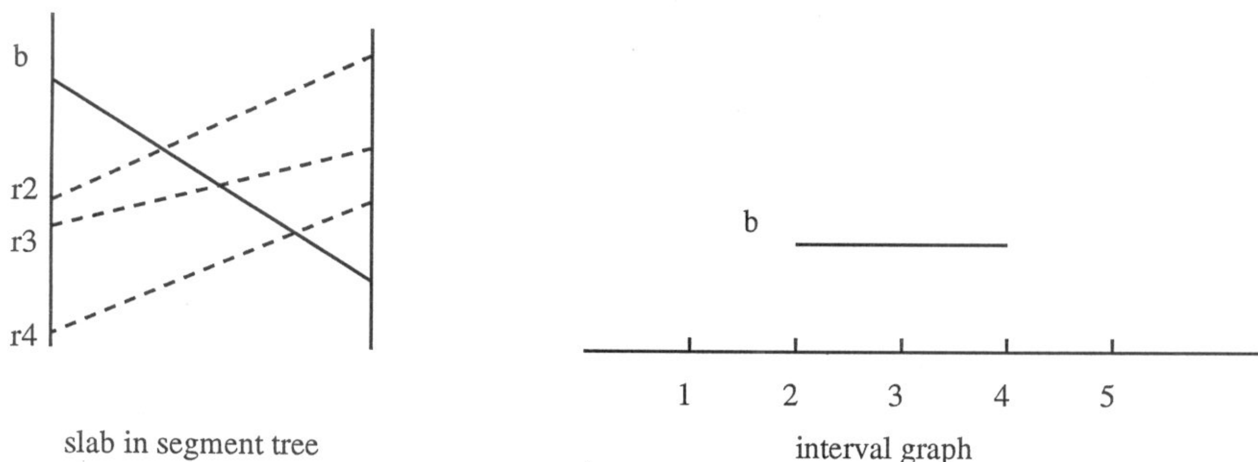

Figure 2.2:

segment and any segment in the event list. If we encounter a right end point of blue segment, we delete it from the event list. As we move to the right, every time we cross a grid point, we do UNION-FIND on the red segment whose index is the horizontal value of the grid point with any segment on the event list.

2.3 Analysis

Let the total number of blue segments be m and that of red segments be n, and $N = m + n$. thus

Step 1 takes $O(N)$ time.

Step 2 takes $O(N \log N)$ time.

The time cost at each node μ is $O((m_\mu + n_\mu^*) \log m_\mu + (n_\mu + m_\mu^*) \log n_\mu)$ process time and $O(m_\mu + n_\mu + m_\mu^* + n_\mu^*)$ UNION-FIND merge time. Since the total size of the segment tree is $O(N \log N)$, the merge time is $O(N \log N)$. The total process time is at most a factor of $O(\log N)$ over the total size of the tree. So this algorithm works in time $O(N \log^2 N)$ and memory space $O(N \log N)$.

2.4 Correctness

First we will show by induction that if two line segments are in the same connected component, the algorithm will report the correct result.

1. If the red line segment and blue segment intersect each other, then the intersection point will fall in one slab in the segment tree. The intersection should be either between two long segments or a long segment and a short segment. We don't need to consider the case where the intersection is between two short segment since it will be taken care of further down the tree.

Suppose the intersection is between blue long segment b and red long segment r. In step 3.3, we assume that b intersects with red long segments $r_{min(i,j)+1}, \ldots, r_{max(i,j)}$. Then r must be among the red long segments. In the interval graph, b spans the interval $[min(i,j)+1, max(i,j)]$ and the index of r must be the horizontal value of a grid point in that interval. According to step 3.4, b and r will be merged together by UNION-FIND operation, if b is the only segment in the event list when we sweep across r. If b is not the only segment in the event list, then r intersects more than one segments in this slab. By merging r with any segment in the event list, r will eventually be merged with b since all segment in the event list are merged together.

Similarly, we can show that the algorithm reports the correct result when the intersection is between a long segment and a short segment.

2. If two line segments are connected through a sequence of intersections, by 1, each consecutive pair will be in the same connected component. So they all belong to the same set at the end of the algorithm.

It is straight forward to show that if the algorithm merges two line segments together, they indeed belong to the same connected component.

2.5 Speeding up the Algorithm

First we show how to reduce the storage requirements. Observe that we don't need to maintain the entire segment tree, but only process it node by node. We therefore traverse the tree in preorder so that at any time we maintain only a single path of $\mathcal{T}$. As proved in [CEGS], the total space at any time is $O(N)$.

To cut down the time from $O(N \log^2 N)$ to $O(N \log N)$ we need to achieve two goals. One is to avoid having to sort the standard lists at each node of segment tree. The other is to speed up the search of elements of hereditary lists in the standard lists. The operation is shown in [CEGS]. We will briefly describe the ideas below.

To achieve the first goal, we begin by finding a linear extension of the following relation on the blue line segments: $b \prec b'$ if the x-projections of b and b' overlap and b' lies above b along a vertical line that intersects both. The relation can be computed in time $O(m \log m)$ by a simple left to right sweep, and a linear extension can then be found in time $O(m)$. We sort R in a similar manner. If we always maintain this order when we generate all the lists B_μ and R_μ from their parent hereditary lists, we will automatically get all these lists sorted. The total time consumed by this procedure is proportional to the total size of all lists which is $O(N \log N)$.

The second goal of speeding up the binary search can be achieved by *fractionally cascading* of the the standard lists (see [CG] for a complete description of the data structuring technique). As we go down the current path we maintain each standard list in a padded-up form, so that it also contains some elements of the standard lists of ancestor nodes along the path. As we go down from node k to one of its children v, we take every fourth element of the (padded) list B_k, pass down these elements to v and merge them with B_v. Similar action is performed on R_k and R_v. These operations neither increase the storage nor the time of the algorithm by more than a constant factor.

When we back up from v to k, we can assume inductively that each endpoint of any red line

segment $r \in R_v^* \cup R_v$ has been located among the line segments in B_v. Since this list also contains a portion of the list B_k, it can be done in an additional constant time per endpoint, by maintaining appropriate pointers between these lists, to locate all these endpoints among the line segments in B_k as well. Same speedup can be achieved symmetrically for R_k. Notice that we thus construct the lists in preorder and evaluate them in an order which is similar to postorder. More specifically, the evaluation at a node k is done in two steps, once when we back up from its left child and then again when the recursion returns from its right child.

Therefore the connected component problem for N bichromatic line segments in the plane can be solved in $O(N \log N)$ time and $O(N)$ space.

3 Bichromatic Simple Polygons

Given blue simple polygons with total m sides, and also red simple polygons of total n sides in the plane, we will show how to solve the connected component problem in this case in $O(N \log N)$ time and $O(N)$ space, where $N = m + n$.

Lemma 3.1 Given n by m bichromatic triangles, their connected components can be determined in $O(N \log N)$ time and $O(N)$ space, where $N = n + m$.

Proof. We prove the lemma by presenting an algorithm. Recall that two triangles intersect if, case (1), they intersect on the boundary or, case (2), one encloses the other.

Case (1) Initially each connected component contains 3 edges of one triangle. We color all the edges of blue triangles blue, and all edges of red triangles red. Then run the connected component algorithm described in section 2 on these $3N$ blue and red line segments. The case where same colored edges that share a common endpoint can be taken care of by simply ignoring the endpoint.

Case (2) Since the enclosure level is at most 1, i.e. there isn't any red triangle that is enclosed in a blue triangle which is also enclosed in a red triangle, and vice versa, we can do 2 left to right line sweep to detect enclosures. First detect the case where blue triangles encloses red ones, and then the other case symmetrically.

We show how to detect enclosure of red triangles by blue one. We do a plane sweep from left to right through the triangles. The schedule is a sorted list of all vertices of blue triangles and the left most vertex of each red triangles in increasing x-order. The cross-section of sweeping line is a sorted list of blue triangles, defined by their upper and lower edges. If the vertex encountered by the sweeping line is a left most vertex of a red triangle, we do a binary search to locate the blue triangle it falls in. If there is such a triangle then check if the red triangle is enclosed by that blue one in $O(1)$ time. Merge them together if the answer is yes. In the case that a red triangle only has a vertical left most edge, then we use that edge to do a range search to locate if there is a single blue range on sweep line that encloses this red triangle. Then proceed as above. If the current event is a vertex of a blue triangle, we either add, delete, or modify the range of the blue triangle according to whether the vertex is the left most, right most, or middle vertex, respectively.

The total running time is $O(N \log N)$ and space is $O(N)$. □

We can extend the above algorithm to prove the following theorem

Theorem 3.2 Given a set of bichromatic simple polygons with N total sides, the connected component problem can be solved in $O(N \log N)$ time and linear space.

Proof. First we triangulate each simple polygon in total $O(N)$ time. Color all these triangles that belong to blue polygons blue and those belong to red polygons red. Initially each connected component contains all the edges (original and new) of one polygon. Then run the algorithm for bichromatic triangles on these triangles. □

Acknowledgments. The authors would like to thank Weiping Shi and Herbert Edelsbrunner for encouragement and many helpful suggestions and discussions throughout the research.

References

[CE] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Proc. 29th IEEE Sympos. Foundations Comput. Sci.*, 1988, 590–600.

[CEGS] B. Chazelle, H. Edelsbrunner, L. J. Guibas and M. Sharir. Algorithms for bichromatic line segment problems and polyhedral terrains. Manuscript.

[CG] B. Chazelle and L. J. Guibas. Fractional cascading: 1. A data structuring technique. *Algorithmica* 1 (1986), 163–191.

[ELOW] H. Edelsbrunner, J. van Leeuwen, T. Ottmann and D. Wood. Computing the connected components of simple rectlinear geometrical objects in d space. *RAIRO Inform. Theor.* 18 (1984), 171–183.

[GOS] L. Guibas, Overmars and M. Sharir. SWAP 89.