On Computing Connected Components of Line Segments

Mario Alberto Lopez *Ramakrishna Thurimella**
Department of Mathematics and Computer Science
University of Denver, Denver CO 80208.

Abstract

It is shown that given a set of n line segments, their connected components can be computed in time $O(n^{4/3} \log^3 n)$. A bound of $o(n^{4/3})$ for this problem would imply a similar bound for detecting, for a given set of n points and n lines, whether some point lies on some of the lines. This problem, known as Hopcroft's problem, is believed to have a lower bound of $\Omega(n^{4/3})$. For the special case when the endpoints of each segment fall inside the same face of the arrangement induced by the set of segments, we give a faster algorithm that runs in $O(n \log^3 n)$ time.

1 Introduction

The connected component problem arises during the fabrication phase of VLSI circuits. Two segments s and t are *connected* if and only if there exist segments $s = s_1, s_2, ..., s_k = t$, $k \geq 2$, such that, for $1 \leq i < k$, s_i intersects s_{i+1}. The *connectedness* relation is an equivalence relation and the equivalence classes are called the *connected components*. For example, Figure 1 illustrates a set of 17 segments with 5 connected components.

The connected components correspond to the nets of a VLSI circuit. As net extraction is often performed on many line segments in practice, the computational complexity of finding connected components plays a central role in the implementation of VLSI circuits.

In their paper on computing intersections between two sets of line segments [MS88], Mairson and Stolfi consider the connected component problem as an interesting open problem. They further conjecture that it should be possible to find connected components without computing all intersections. In addition, they provide a reference to a manuscript by Guibas and Sharir [GS87] that contains an $O(n \log^2 n)$ algorithm for computing the connected components of two sets S and T of line segments in the plane, where no two segments in S (similarly, T) intersect. The special case of computing connected components when the objects are orthogonal has been shown to be solvable with $O(n)$ space in $O(n \log n)$ time, where n is the size of the input [ELOW84, GS83]. When the number of orientations of the input segments is c, it is known that the connected components can be found in $O(cn \log n)$ time with $O(n)$ space [LJS91]. The number orientations can be arbitrary in today's technology, and hence many practical net extraction algorithms compute all intersections [NS88]. This approach can be rather expensive computationally as there can be $\Theta(n^2)$ intersections for a set of n objects.

*This author's research was supported in part by National Science Foundation grant CCR-9210604.

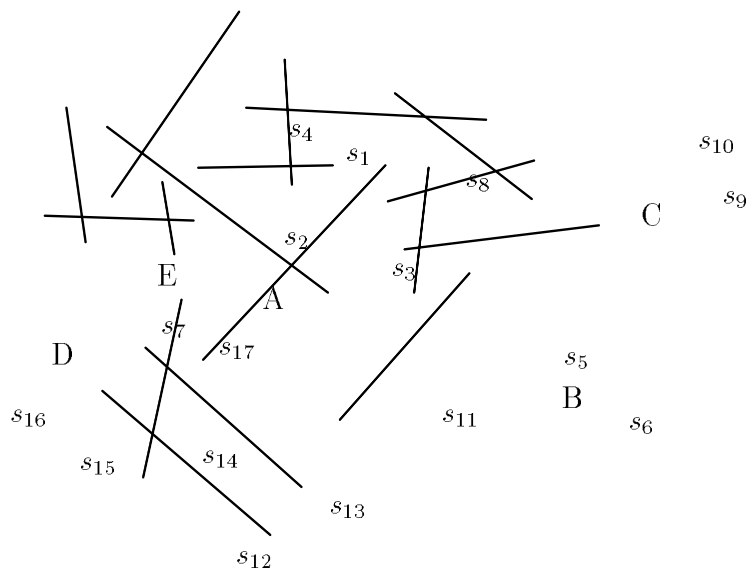

Figure 1: A set of line segments and their connected components.

The main result of this paper is an algorithm to compute connected components in $O(n^{4/3}\log^3 n)$ time with $O(n^{4/3})$ space. Computing connected components is intimately related to Hopcroft's problem — given n lines and n points in the plane, deciding whether some point lies on some of the lines. It is believed that this problem has a lower bound of $\Omega(n^{4/3})$. We quote from a recent paper of Matoušek [Ma92], in which an algorithm with a time bound of $n^{4/3}2^{O(\log^* n)}$ is given for Hopcroft's problem:

> "Understanding this problem seems to be one of the major challenges in computational geometry ... It is suspected that $n^{4/3}$ might be the true computational complexity of this problem, although nothing even approaching a proof is known."

Hopcroft's problem can be easily reduced to the connected component problem: find a rectangle $\mathcal{R}$ that intersects every input line and whose interior contains all the points, clip the lines using $\mathcal{R}$ and discard the portions that lie outside $\mathcal{R}$, and add the sides of $\mathcal{R}$ to the input (thus making one connected component out of all line segments). For this input, if the number of connected components is less than $(1+n)$, then it can be concluded that there exists a line that goes through a point. Therefore, improving our time bound beyond log factors will have a profound impact and is probably very hard, if at all possible.

We also show how the connected components can be computed in only $O(n\log^3 n)$ time when both endpoints of each segment fall inside the same face; this implies that any proof establishing an $n^{4/3}$ lower bound would have to take into account the way endpoints are distributed among regions. Our algorithm for this special case produces the correct result provided the input is valid. Note that checking the validity of the input is at least as hard as Hopcroft's problem, as the latter problem reduces to the former.

The rest of this paper is organized as follows. The next section gives the necessary background and outlines the algorithm at a high level. Section 3 includes an algorithm for connected components

for the special case when the endpoints of the segments lie on two vertical lines. An algorithm that extends the restricted case to the general case is presented in Section 4. Finally, in Section 5 we establish the time complexity and correctness.

2 Preliminaries

The input is assumed to be a set $S = \{s_1, \ldots, s_n\}$ of line segments. Each segment is described by a pair of (x, y) coordinates denoting its endpoints.

The output of our algorithm is a label for each segment such that two segments get the same label if and only if they are in the same connected component. For example, in Figure 1, the segments s_1, s_4, s_{12}, and s_{13} belong to the connected component labeled E.

Our algorithm makes use of a well known duality transformation (for an example see [Ch86]). Under this transform, a point (a, b) is mapped to the line $y = ax + b$; and a line $y = ax + b$ is mapped to the point $(-a, b)$. A point p lies above (resp., below) a line L if and only if the dual of L lies below (resp., above) the dual of p. Similarly, a segment s is mapped to a double wedge: the intersection of two pairs of halfplanes obtained by mapping the endpoints of s to their dual representation. The inside of the wedge is just the locus of duals of all lines that intersect the segment s. In other words, a line L intersects a segment s if and only if the dual of L is inside the dual of s. For an illustration see Figure 2.

Figure 2: Dual transformation of a segment s and a line L.

The key data structure in our algorithm is the segment tree. For the sake of completeness, we include its definition (taken from [PS88]) in the following.

The *segment tree* is a rooted binary tree designed to store intervals on the real line whose extremes belong to a fixed set of N abscissae. The abscissae can be normalized by replacing each of them by its rank in their left-to-right order, i.e., the abscissae can be considered to be the integers in the range $[1, N]$. For a given pair of integers l and r, $l < r$, the segment tree $T(l, r)$ is recursively built as follows: It consists of a root v, with parameters $B[v] = l$ and $E[v] = r$, and if $r - l > 1$, of a left subtree $T(l, \lfloor B[v] + E[v]/2 \rfloor)$ and a right subtree $T(\lfloor B(v) + E(v)/2 \rfloor, r)$. The

roots of these subtrees are the *left* and *right* children of v. The parameters $B[v]$ and $E[v]$ define the *interval*$[B[v], E[v]] \subseteq [l, r]$. The intervals associated with internal (resp., leaf) nodes are *canonical* (resp., *elementary*) *intervals*.

It is easy to show that $T(l, r)$ is balanced and has depth $\lceil \log_2(r - l) \rceil$. This data structure is designed to store intervals whose extremes belong to the set $\{l, l+1, ..., r\}$. It is straightforward to show that for $r - l > 3$, an arbitrary interval $[b, e]$, with integers $b < e$, can be partitioned into a collection of at most $\lceil \log_2(r-l) \rceil + \lfloor \log_2(r-l) \rfloor - 2$ intervals of $T(l, r)$. Basically, $[b, e]$ is stored at each node v such that $[B[v], E[v]] \subseteq [b, e]$ but $[B[v'], E[v']] \not\subseteq [b, e]$, where v' is the parent of v. For more details on the segment tree data structure, the reader is referred to [PS88].

We now outline our strategy:

1. Build a segment tree T on the $2n$ x-coordinates of the input endpoints. Partition the input segments using their projections onto the x-axis and store the resulting pieces in T.
2. Compute the connected components of line segments associated with each node of the segment tree.
3. Starting at the level one above the leaves and finishing at the root, for each node v at that level, merge the connected components resulting from the descendants of v with the connected components at v.

We represent the partially computed connected components using the union-find data structure given in [AHU74], where each *find* takes $O(1)$ time and a sequence of $n - 1$ *union* operations takes $O(n \log n)$ time in worst case. The partial component containing an arbitrary segment is found by doing a find operation and two partial components are merged by doing a union operation. Clearly, the algorithm will perform at most $n - 1$ unions, but the number of finds can be $O(n \log^2 n)$, as shown later.

Notice that the segment tree partitions each input segment into $O(\log n)$ pieces. From the original input of size n, it can be assumed that we construct another input which is a set of $O(n \log n)$ segments. Initially, all pieces of a segment are placed in the same partial component. Henceforth, a segment refers to a piece resulting at the time when we construct the new input.

It might help to visualize each node v of T as being "responsible" for the segments stored at either v or one of the descendants of v. At the end, the root node contains the required output.

The main difficulty posed by this problem is Step 3 of the outline above.

3 The case when endpoints lie on an interval

We now present an efficient algorithm to compute the connected components of a set of segments each of whose endpoints lie one on each of two lines L and R parallel to the y-axis. Refer to Figure 3 for an example.

Algorithm *CC on an interval*

1. Sort the segments in non-decreasing order by the y-coordinate of the endpoints on line L. Let the sorted y-coordinates on L be $l_1, l_2, ..., l_k$. Also, let the corresponding y-coordinates on line R be $r_1, r_2, ..., r_k$. Break ties $l_i = l_j$ by forcing $i < j$ if $r_i > r_j$.

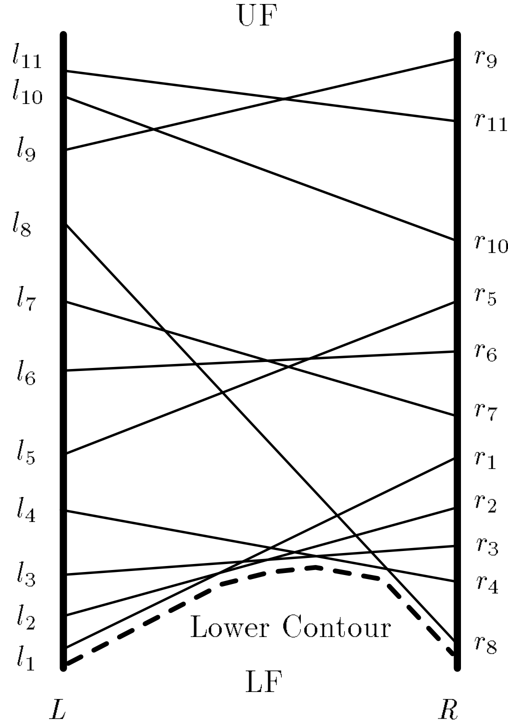

Figure 3: An example of line segments between an interval.

2. Push r_1 onto a stack. Now, sweep L starting at l_2: for each segment at l_i, $i > 1$, compare r_i with the top of stack t and push r_i onto the stack if $r_i > t$; else, pop the stack until the top of the stack is less than r_i. For every popped element r_j, merge the connected components of r_j and r_i. Push back the previous top of the stack, i.e. t, onto the stack to represent the connected component just formed.

Theorem 1 *The connected components of k line segments in an interval can be computed in $O(k \log k)$ time and $O(k)$ space.*

Proof: Apply the above algorithm. We prove correctness by showing that all and only intersecting segments are placed in the same partial component (at some time). Since partial components are never split, this implies the desired result. For simplicity assume no two segments share the same left or right endpoint. The case where segments share an endpoint can be handled easily (by suitably modifying the sorting procedure). Note that at all times the stack stores one representative element for each partial component computed so far. This representative is the y-coordinate of the highest right endpoint of all segments in the same partial component. Consider segments s_i and s_j such that $l_i > l_j$, i.e., s_i is processed after s_j. If during the stack popping phase we find that $r_i < r_j$ then clearly s_i and s_j intersect and thus they are correctly placed in the same component. We now show that all pairs of intersecting segments are eventually placed in the same component.

Suppose s_i intersects s_j, i.e., $l_i > l_j$ and $r_i < r_j$. Let r_h be the stack representative of the partial component of s_j at the time s_i is processed. Since $l_i > l_h$ and $r_i < r_j < r_h$, s_i and s_h (and hence, s_j) are placed in the same component and the claim follows. The space required is clearly $O(k)$ since the stack will not store more than k endpoints. Sorting the segments by left endpoint requires $O(k \log k)$ time. Since the total number of pop operations cannot exceed k (and each such pop results in a union), each endpoint can be processed in $O(\log k)$ amortized time. 2

4 Merging

In this section, we consider the merging step at an arbitrary internal node v of the segment tree. We examine each segment s originating from a descendant of v and determine the connected components at v intersected by s[†].

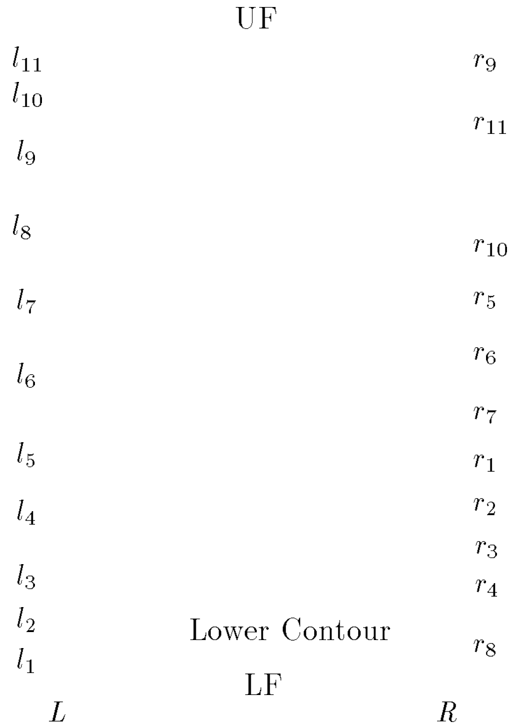

Figure 4: The contours of the connected components of Figure 3 are totally ordered.

Our merging algorithm partitions the set of segments originating from the descendants depending on whether a segment is completely contained within the boundaries of a connected component at v or not, and then merges each segment separately with the components found at v. We now formalize our strategy.

[†]A segment s intersects a connected component C if and only if s intersects any segment of C.

Associated with each node v of T there are infinite vertical lines L and R, at $B[v]$ and $E[v]$, respectively. Consider a connected component C at v. Each segments of C has one endpoint on L and one on R. The two vertical lines together with C induce a partition of the plane into faces, vertices and edges (known in the literature as an *arrangement*). In this partition of the plane, there are exactly two unbounded faces enclosed between L and R. We denote by UF (resp. LF) the face unbounded in the upward (resp. downward) direction, and refer to its edges as the *upper* (resp. *lower*) contour of C. See Figure 3 for an example.

All faces in an arrangement, particularly UF and LF, are convex [EGS90]. Now consider the upper and lower contours of a set of connected components at node v. Observe that these contours can be totally ordered by considering, for each contour, the y-coordinate of its vertex on L. See Figure 4 for an example.

Algorithm *Merge at v*

1. Build the lower and upper contours of the connected components computed at v.
2. Partition the descendants of v into *local* and *global* segments: A segment is said to be *local* if both of its endpoints lie within the upper and lower contour of a single connected component at v. A segment is *global* if it intersects at least one contour at v. Segments that are neither local nor global intersect no segments at v and can be ignored.
3. For each segment s from a strict descendant of v do the following:

   Case 1 (s is local) : Let C be the connected component at v whose lower and upper contours enclose s. If s intersects any segments from C, merge C with the connected component of s.

   Case 2 (s is global) : Let C be a connected component at v such that s intersects at least one of the contours of C. For each such C, merge C with the partial component of s.

Figure 5 shows for a vertex v, the set of all segments that are the responsibility of v. The connected components at v are shown in bold. The rest of the segments are those that belong to the descendants of v. The solid lines that are not bold are the segments that belong to the immediate children of v in T. Segment s_1 is local to C whereas s_2 is not.

In the rest of this section, we give details and analyze complexity of each step of *Merge at v*.

4.1 Classify segments into locals and globals

This section describes the classification of segments as well as an algorithm for computing the contours of a connected component consisting of k segments. We only include the description for finding the lower contour; a similar algorithm can easily be derived for the upper contour. Note that each segment can contribute at most one edge in a contour. The output of the algorithm is the list of vertices of the lower contour stored in a stack LC of size at most k (from $LC[1]$ to $LC[top]$) in increasing order of x-coordinate. Segments are processed in ascending order by intersection with L. An invariant of the algorithm is that LC always contains the lower contour of the segments processed so far. A new segment is either entirely above the current contour or it cuts across it.

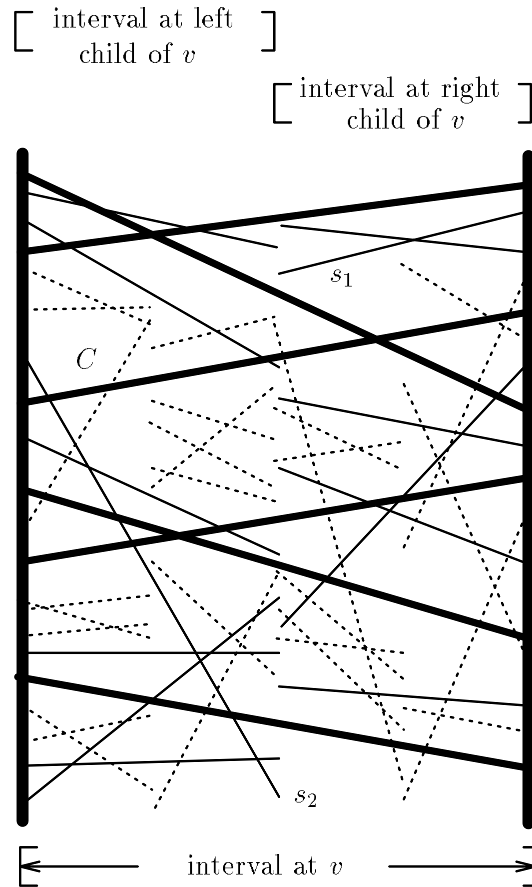

Figure 5: Merging step at node v.

In the latter case, vertices are popped from LC in a manner reminiscent of vertex deletion during a Graham scan [PS88].

Algorithm *Lower Contour*

1. Sort the y-coordinates of the endpoints of the segments on L in ascending order; if two or more segments share an endpoint, keep only the one with the smallest slope. Let the points in sorted order be $l_1, l_2, ..., l_k$. Denote the segment that contributed $l_i, 1 \leq i \leq k$, by s_i. Also, let the y-coordinates on line R of $s_1, s_2, ..., s_k$ be $r_1, r_2, ..., r_k$.

2. Push l_1 and r_1 (the endpoints of s_1) onto LC. Now sweep L starting at the bottom: for every point l_i on L, $2 \leq i \leq k$, if the stack top is below s_i discard s_i; otherwise pop the stack until s_i intersects the segment s from $LC[top]$ to $LC[top-1]$. Pop $LC[top]$. Push the intersection of s_i and s. Also push r_i, the right endpoint of s_i, onto LC.

Lemma 1 *Algorithm* Lower Contour *correctly computes the lower contour of a connected component consisting of k segments. Moreover, it can be implemented to run in $O(k \log k)$ time ($O(k)$ if the endpoints on L are already sorted).*

Proof: We prove correctness by induction on k. The case where $k = 1$ is trivial, as the contour is just s_1. Assume the contour of the first $m-1$ segments has been computed correctly. Let q_a, where $1 < a \leq top$, denote the segment from C that contributes the contour segment from $LC[a-1]$ to $LC[a]$. Since the contour is convex, $a < b$ implies that $slope(q_a) > slope(q_b)$ and $l_a < l_b$. If s_m changes the current contour, then $slope(s_m) < slope(q_{top})$, $l_m > l_{top}$, and s_m crosses the contour exactly once, say between $LC[h-1]$ and $LC[h]$. All points $LC[h] \ldots LC[top]$ are above s_m and should be removed from the stack. The new contour is thus correctly computed in Step 2. Sorting the left endpoints requires $O(k \log k)$ time. Since no segment can contribute more than one contour segment and a contour segment can be popped at most once, processing an endpoint requires $O(1)$ amortized time. 2

Once the contours are computed, it is a simple matter to classify segments as local and global.

Lemma 2 *A set of d_v segments can be partitioned into local and global with respect to a set of connected components consisting of n_v segments at v in $O(n_v + d_v \log n_v)$ time.*

Proof: In order to establish the time bounds, we use the planar point location algorithm from [EGS86]. Consider the contours of the connected components at v without the edges from L and R. These chains induce a monotone subdivision of the plane when we attach infinite horizontal extensions to their ends. Now, we identify the local and global segments after locating their endpoints. This takes $O(\log n_v)$ per segment for a total of $O(n_v + d_v \log n_v)$, as claimed. 2

4.2 Merge local segments

Assume we are given a connected component C at v consisting of k segments with endpoints on the interval associated with v, and a set of h segments whose endpoints lie within the region bounded by the interval and the upper and lower contours of C. We show that C can be merged with its local segments in time $O(m^{4/3} \log^{2/3} m)$ where $m = h + k$.

The technique used was originally suggested by Agarwal [Ag90a] and later refined in [Ma91]. We borrow the notation of [Ag90b].

Assume the segments at v are colored blue and the local segments are colored red. We are interested in detecting red-blue intersections.

Algorithm *Merge-locals*

1. Fix $r = m^{1/3} \log^{-1/3} m$ and partition the plane into $O(r^2)$ triangles so that each triangle is intersected by no more than $O(m/r)$ segment supporting lines and contains $O(m/r^2)$ segment endpoints. We say that a segment is *short* with respect to a triangle if it has at least one endpoint inside the triangle. Otherwise, a segment that intersects the triangle is said to be *long*.

2. Process red-blue intersections within each triangle as follows:

(a) *Long-long* intersections: For each red segment, if the number of intersections with blue segments is nonzero, then merge the connected component of the red segment with C.

(b) *Short-long* intersections: Find the wedges corresponding to red short segments by applying the duality transformation of Section 2. Construct the arrangement induced by these wedges in the plane. Associate with each region a list of the wedges which contain that region. Observe that the supporting line of each blue long segment becomes a point in the dual. For each such point, find the region containing it and mark it. Finally, for each marked region, traverse the associated list of wedges and merge the corresponding short red segments with C.

(c) Note that extending the blue segments does not increase the number of red-blue intersections. Consequently, we disregard *short-short* intersections.

Lemma 3 *Algorithm* Merge-locals *takes* $O(m^{4/3} \log^{2/3} m)$ *time where* m *is the sum of the number of segments in* C *and the number of segments local to* C.

Proof: Let us estimate the time complexity of the above steps. Step 1 takes $O(mr)$ time [Ma91]. (Actually, [Ma91] does not show how to ensure that each triangle contains no more than $O(m/r^2)$ endpoints. However, this additional requirement can be satisfied with no increase in complexity as shown in Section 4 of [Ag90b].) Consider an arbitrary triangle from Step 1. Let n_s and n_l denote the number of short and long segments, respectively. Clearly $n_l = O(m/r)$ and $n_s = O(m/r^2)$. Step 2a can be done in $O(n_l \log n_l)$ time [Ag90b]. Constructing the arrangement and finding the list of wedges for each region can be done in $O(n_s^2)$ time [EOS86]. Point location, which takes $O(\log n_s)$ time per query using [EGS86], is performed for n_l points, resulting in $O(n_l \log n_s)$ time. Thus, Step 2b requires $O(n_s^2 + n_l \log n_s)$ time. The total time for Step 2 is $O(r^2(n_l \log n_l + n_s^2 + n_l \log n_s))$. Eliminating n_l and n_s and simplifying we get a bound of $O(mr \log m + m^2/r^2 + mr \log(m/r^2))$. Substituting for r yields the desired bound. 2

4.3 Merge global segments

Merging global segments is a straightforward application of the planar point location algorithm of [EGS86] along with some union operations. Let $C_1, \ldots, C_k$ be the connected components at v sorted in ascending order by the intersection of L with their lower contours (see Figure 4). With each global segment s, we associate an interval $[i, j]$, where C_i (resp., C_j) is the first (resp. last) connected component in $C_1, \ldots, C_k$ that has an intersection with s.

Algorithm *Merge-globals*

1. For each global segment s, find the interval associated with s.

2. Find the union of the intervals associated with the global segments.

3. If $[a, b]$ is an interval resulting from the previous step after the union, then merge the connected components $C_a, C_{a+1}, \ldots, C_b$.

Lemma 4 *Let* n_v *and* d_v*, respectively, be the number of segments and global segments at* v*. The connected components at* v *can be merged using the global segments in* $O((n_v + d_v) \log n_v)$ *time.*

Proof: We only analyze the time complexity as the correctness is obvious. Step 1 can be implemented while classifying segments into local or global (see Lemma 2). For each global segment s with endpoints p and q such that the y-coordinate of p smaller than that of q, we perform point location for p and q. If p (resp., q) belongs to a region R that is the interior of a connected component C_k (resp., C_l), then the first (resp., second) component of the interval associated with s is k (resp., l); otherwise, the first (resp., second) component of the interval associated with s is C_k (resp., C_l) where the interior of C_k (resp., C_l) is the region immediately above (resp., below) R. Modifying the data structure of [EGS86] to keep the neighboring regions is trivial. Therefore, Step 1 takes $O(d_v \log n_v)$ time. Step 2 can be implemented by first sorting the interval endpoints and then scanning them in the increasing order. This requires at most $O(n_v \log n_v)$ operations. Step 3 requires at most n_v union-find operations. 2

5 Complexity and Correctness

We estimate the overall time complexity and establish correctness by proving the following main theorem:

Theorem 2 *The connected components of n line segments can be computed in time $O(n^{4/3} \log^3 n)$.*

Proof: First we argue correctness. Clearly the algorithm only merges connected components of intersecting segments. We need to show that every pair of intersecting segments ends up in the same component. Consider segments p and q that intersect at point I. Let u (resp., v) be the node of T that stores the subsegment of p (resp., q) that contains I. If $u = v$, after computing the connected components at u, p and q are in the same component. This follows from Theorem 1. Otherwise, assume w.l.o.g. that v is a descendant of u. The subsegment of q at v becomes a local or global segment with respect to u. Correctness follows from Lemma 3.

Building the segment tree T requires $O(n \log n)$ time. Let n_v denote the number of segments at node v. The total number of segments in T is $\sum_{v \in T} n_v = O(n \log n)$. By Theorem 1, computing the connected components at individual nodes requires $\sum_{v \in T} O(n_v \log n_v) = O(n \log^2 n)$ time.

We now analyze the complexity of algorithm *Merge at v*. Let d_v the number of segments originating from the descendants of v. By Lemma 1, Step 1 requires $O(n_v)$ time. Step 2 takes $O(d_v \log n_v)$ time as argued in Lemma 2. Merging local segments according Lemma 3 can be done in $O((d_v + n_v)^{4/3} \log^{2/3}(n_v + d_v))$ time. Finally, in $O((n_v + d_v) \log n_v)$ time the globals can be merged. Consider V, the set of nodes belonging to an arbitrary level of T. Since $\sum_{v \in V} n_v \leq n$ and $\sum_{v \in V} d_v \leq n \log n$, the total time spent *per level* of T is

$$O(\sum_{v \in V} n_v + d_v \log n_v + (n_v + d_v)^{4/3} \log^{2/3}(n_v + d_v) + (n_v + d_v) \log n_v) = O(n \log^2 n + n^{4/3} \log^2 n)$$

Since T has $O(\log n)$ levels, the total complexity of the algorithm is $O(n \log^3 n + n^{4/3} \log^3 n) = O(n^{4/3} \log^3 n)$, as claimed. 2

5.1 Handling vertical segments

Vertical segments cannot be handled by the previous approach because such segments do not span any of the elementary intervals associated with the segment tree T. This problem can be solved

by rotating the set of input segments so that no segment is vertical. A more practical approach is to treat each vertical segment s as "a descendant segment" of any node v of T whose canonical or elementary interval contains s (at most two paths in T). If s is a global segment with respect to v, it is handled in the same way as other global segments. If s is local, then the dual of s becomes the locus of points between two parallel lines.

$\mathcal{A}$

(a) (b)

Figure 6: (a) A connected component and (b) the arrangement $\mathcal{A}$ induced by the zones of L and R.

5.2 Segments whose endpoints belong to the same face

The set S of segments induces a partition of the plane into a number of faces (an arrangement). The case where both endpoints of each segment fall strictly inside the same face (see Figure 1 for an example) can be solved in $O(n \log^3 n)$. Most of the algorithm remains the same, except for the merging of locals which is now simplified. Let C be a connected component with k segments at some node v. Consider the zone of L, i.e., the the faces of C containing an edge whose supporting line is L. This zone contains $O(k)$ edges [AMS91] and can be computed in $O(k \log^2 k)$ time [Mi90]. The same bounds apply when computing the zone of R. Let $\mathcal{A}$ be the arrangement induced by the zones of L and R (see Figure 6b). Consider each segment s local to C. Merge s with C if the endpoints of s belong to different faces of $\mathcal{A}$. Clearly, this procedure does not create false intersections. To show that no intersections are missed it suffices to notice that every input segment that intersects C must intersect a zone edge. Hence, one of its pieces in the segment tree will have endpoints on different faces of $\mathcal{A}$. The time complexity for merging locals, and hence of the whole algorithm, becomes $O(n \log^3 n)$. Note that this algorithm produces the correct result provided the input is valid. Checking the validity of the input is at least as hard as Hopcroft's problem, as the latter problem reduces to the former.

Acknowledgments. We wish to thank the anonymous referee who suggested an alternate way

of merging local segments, shown in Section 4.3, that resulted in an improvement of a $O(\log^{1/2} n)$ factor in the overall complexity of the algorithm.

References

[Ag90a] P.K. Agarwal, *Partitioning arrangements of lines I: An efficient deterministic algorithm*, Discrete Comput. Geom., 5 (1990), pp. 449-483.

[Ag90b] P.K. Agarwal, *Partitioning arrangements of lines II: applications*, Discrete Comput. Geom., 5 (1990), pp. 533-573.

[AHU74] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[AMS91] B. Aronov, J. Matoušek and M. Sharir, *On the sum of squares of cell complexities in hyperplane arrangements*, Proc. 7th Annual ACM Symp. Comput. Geom., (1991), pp. 307-313.

[Ch86] B. Chazelle, *Filtering search: A new approach to query-answering*, SIAM J. Comput., 15:3 (1986), pp. 703-724.

[EGS86] H. Edelsbrunner, L.J. Guibas and J. Stolfi, *Optimal point location in a monotone subdivision*, SIAM J. Comput., 15:2 (1986), pp. 317-340.

[EGS90] H. Edelsbrunner, L.J. Guibas and M. Sharir, *The complexity and construction of many faces in an arrangement of lines and of segments*, Discrete Comput. Geom., 5 (1990), pp. 161-196.

[EOS86] H. Edelsbrunner, J. O'Rourke and R. Seidel, *Constructing arrangements of lines and hyperplanes with applications*, SIAM J. Comput., 15:2 (1986), pp. 341-363.

[ELOW84] H. Edelsbrunner, J. van Leeuwen, T. Ottmann and D. Wood, *Computing connected components of simple rectilinear geometrical objects in d-space*, RAIRO Theoretical Informatics, 18:2 (1984), pp. 171-183.

[GS83] L.J. Guibas and J. Saxe, *Problem 80-15*, Journal of Algorithms, 4 (1983), pp. 176-181.

[GS87] L.J. Guibas and M. Sharir, *Computing the unbounded component of an arrangement of line segments*, Manuscript, (1987).

[LJS91] M. Lopez, R. Janardan and S. Sahni, *A fast algorithm for VLSI net extraction*, submitted.

[Mi90] J.S.B. Mitchell, *On computing a single face in an arrangement of line segments*, Manuscript, (1990).

[MS88] H.G. Mairson and J. Stolfi, *Reporting and counting intersections between two sets of line segments*, NATO ASI series, Theoretical foundations of computer graphics and CAD, Springer-Verlag (1988), pp. 307-325.

[Ma91] J. Matoušek, *Cutting hyperplane arrangements,* Discrete Comput. Geom., 6 (1991), pp. 385-406.

[Ma92] J. Matoušek, *Range searching with efficient hierarchical cuttings*, Proc. 8th Annual ACM Symp. Comput. Geom., (1992), pp. 276-285.

[NS88] S. Nahar and S. Sahni, *Time and space efficient net extractor*, Computer Aided Design, 20:1 (1988), pp. 17-26.

[PS88] F. Preparata and M.I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag, 1988.