Discrete Comput Geom 4:433-466 (1989)

Discrete & Computational
Geometry

© 1989 Springer-Verlag New York Inc

Implicitly Representing Arrangements of Lines or Segments*

Herbert Edelsbrunner,[1] Leonidas Guibas,[2,3] John Hershberger,[3]
Raimund Seidel,[4,5] Micha Sharir,[6,7] Jack Snoeyink,[2]
and Emo Welzl[8]

[1] University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA

[2] Stanford University, Stanford, CA 94305, USA

[3] DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA

[4] IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, USA

[5] University of California at Berkeley, Berkeley, CA 94720, USA

[6] New York University, 251 Mercer Street, New York, NY 10012, USA

[7] Tel Aviv University, 69978 Tel Aviv, Israel

[8] Freie Universität Berlin, Arnimallee 2-6, 1000 Berlin 33, Federal Republic of Germany

Abstract. An *arrangement* of n lines (or line segments) in the plane is the partition of the plane defined by these objects. Such an arrangement consists of $O(n^2)$ regions, called *faces*. In this paper we study the problem of calculating and storing arrangements *implicitly*, using subquadratic space and preprocessing, so that, given any query point p, we can calculate efficiently the face containing p. First, we consider the case of lines and show that with $\Lambda(n)$ space[1] and $\Lambda(n^{3/2})$ preprocessing time, we can answer face queries in $\Lambda(\sqrt{n}) + O(K)$ time, where K is the output size. (The query time is achieved with high probability.) In the process, we solve three interesting subproblems: (1) given a set of n points, find a straight-edge spanning tree of these points such that any line intersects only a few edges of the tree, (2)

---

* The first author is pleased to acknowledge the support of Amoco Fnd. Fac. Dev. Comput. Sci. 1-6-44862 and National Science Foundation Grant CCR-8714565. Work on this paper by the fifth author has been supported by Office of Naval Research Grant N00014-87-K-0129, by National Science Foundation Grant NSF-DCR-83-20085, by grants from the Digital Equipment Corporation, and the IBM Corporation, and by a research grant from the NCRD—the Israeli National Council for Research and Development. The sixth author was supported in part by a National Science Foundation Graduate Fellowship. This work was begun while the non-DEC authors were visiting at the DEC Systems Research Center.

[1] We use $\Lambda(f(n))$ to abbreviate $O(f(n) \log^k n)$, where k is a constant.

given a simple polygonal path Γ, form a data structure from which we can find the convex hull of any subpath of Γ quickly, and (3) given a set of points, organize them so that the convex hull of their subset lying above a query line can be found quickly. Second, using random sampling, we give a tradeoff between increasing space and decreasing query time. Third, we extend our structure to report faces in an arrangement of line segments in $\Lambda(n^{1/3})+O(K)$ time, given $\Lambda(n^{4/3})$ space and $\Lambda(n^{5/3})$ preprocessing time. Lastly, we note that our techniques allow us to compute m faces in an arrangement of n lines in time $\Lambda(m^{2/3}n^{2/3}+n)$, which is nearly optimal.

1. Introduction

Suppose we are given a set L of n straight lines in the plane. These lines partition the plane into a set of convex regions called *faces*. The collection of all these faces is referred to as the *arrangement of* L, and is denoted by $\mathscr{A}(L)$. Now suppose we are allowed to perform some preprocessing on L so that the following type of query can be answered readily: for a given query point p, which face in $\mathscr{A}(L)$ contains p?

What do we mean by "which face"? We want a complete description of the face containing p in terms of the lines in L, that is, which lines of L appear along the boundary of that face in, say, counterclockwise order. In some cases we may be satisfied with a simpler property of the face, such as its intersection with a line, its leftmost vertex, the edge below p, or the number of lines on the face boundary.

One direct approach to this problem is to precompute complete descriptions of all the faces in $\mathscr{A}(L)$. Given a query point p, we then just need to locate p in the face containing it; the precomputed description of that face can be used to answer the query quickly. The complete descriptions of all the faces in $\mathscr{A}(L)$ can be computed in $O(n^2)$ time [CGL], [EOS]; determining which face of $\mathscr{A}(L)$ contains p is an instance of planar point location, which can be solved in $O(\log n)$ time [Ki], [EGS2], [ST] after $O(n^2)$ preprocessing. Thus, after $O(n^2)$ preprocessing, any face-reporting query can be answered in $O(\log n+K)$ time, where K is the "output size," namely the number of lines that bound the desired face. The simpler queries listed above can be answered in $O(\log n)$ time.

But what if it is deemed excessive to use $O(n^2)$ time for preprocessing, or if the $\Theta(n^2)$ space that the above method requires is not available? What if the space at our disposal is only about linear in n?

Again, there is a straightforward solution: do not do any preprocessing at all; simply use any set representation as the data structure for L. When given a query point p, first determine for each line $l\in L$ which of the two halfplanes bounded by l contains p, and then construct the intersection of these halfplanes. This takes $O(n\log n)$ time using divide-and-conquer [SH]. If the lines in L are initially sorted by slope, then the intersection can be computed in $O(n)$ time. This query time compares poorly with the $O(\log n)$ query time achieved by the first method.

In this paper we address the question of whether we can answer face queries in sublinear time using only roughly linear space. We show that it is possible to preprocess a set L of n lines in randomized time $O(n^{3/2} \log^2 n)$, producing a data structure that uses $O(n \log n)$ space, so that, with high probability, any face-reporting query can be answered in time $O(\sqrt{n} \log^5 n + K)$. The simpler queries mentioned above take $O(\sqrt{n} \log^5 n)$ time. (By an alternative approach that uses more precomputation, we can reduce the space to $O(n)$ and guarantee $O(\sqrt{n} \log^3 n + K)$ query time.) We also present a similar result for the more general case when the set L consists of line *segments*, instead of (infinite) lines.[2]

Our solution proceeds roughly as follows: First, using duality, we transform the problem into a related one about data points and query lines.[3] In this dual setting, our problem becomes that of preprocessing a set P of points in the plane so that, for any query line l, we can determine quickly the convex hull of the points in P lying above l, and the convex hull of the points in P lying below l. To achieve this we construct a family of spanning trees for P with the property that any straight line intersects only few edges of one of these trees. This means that any straight line "cuts" such a tree into a small number of pieces, each of which lies completely on one side of the line. Using an appropriate data structure based on the spanning tree, it is possible to derive an implicit representation of the convex hulls of the individual tree pieces quickly, and these individual hulls can then be combined to give the two desired convex hulls.

To ensure the output-size sensitivity of our algorithm, it is imperative that we do not construct the convex hulls of the tree pieces explicitly, because their total size might be larger than the size of the combined hull. We achieve this using a hierarchical representation of convex hulls that is described in Section 4.

Our use of spanning trees is closely related to a recent technique of Chazelle and Welzl [CW] for improved halfplane range searching. However, in our approach we face several additional interesting subproblems; for example, the initial form of the spanning trees that we produce can be highly self-intersecting, which is unsuitable for our application. We develop an efficient procedure for "untangling" such a tree while increasing the maximum number of cuts of that tree by a line by at most a factor of two. The procedure uses the shortest-path algorithm for simple polygons of [GH].

We next consider the case of line segments, which is more complicated because of the highly irregular shape of the faces in an arrangement of segments. We show that by using $\Lambda(n^{5/3})$ randomized preprocessing and $\Lambda(n^{4/3})$ space, we can obtain the face containing a query point in time $\Lambda(n^{1/3}) + O(K)$, where K is the size of the face. Note that in this case we have decreased the query time at the cost of using more preprocessing and space. We show that for the case of lines, this is an instance of a generally applicable tradeoff.

---

[2] Matoušek [M] has recently improved these preprocessing bounds by showing how to build the necessary spanning trees more efficiently. His method runs in $O(n \log^2 n)$ time in the randomized case and in $\Lambda(n^3)$ time in the deterministic case ($\Lambda(n^{2.5})$ arithmetic operations on $\sqrt{n}$-bit numbers). Matoušek also gives an $O(n^{1.4+\delta})$ randomized algorithm (for any $\delta > 0$) that gives a query time of $O(\sqrt{n} \log^4 n + K)$. Matoušek's results improve the preprocessing bounds claimed in Theorems 3.2, 3.5, 5.1, 6.1, and 7.1. For example, the preprocessing needed to represent an arrangement of n line segments (see below) decreases from $\Lambda(n^{5/3})$ to $\Lambda(n^{4/3})$.

[3] We assume the reader is familiar with the concept of point-line duality in the plane. Otherwise consult Brown [B] or Edelsbrunner [E, pp. 12–14].

We believe that our results will have many applications in speeding up algorithms that manipulate arrangements of lines or of segments in the plane. One application, given in this paper, is the computation of m distinct faces in an arrangement of n lines in time $\Lambda(m^{2/3}n^{2/3}+n)$, which, by the results of [CEG*], is almost optimal. Another recent application is the calculation of many components in an arrangement of n (intersecting) triangles in 3-space [AS].

The organization of this paper is as follows: Section 2 describes some of the notation we use. Section 3 gives the spanning-tree construction for the points dual to the lines in L, as well as the algorithm for "untangling" a self-intersecting spanning tree. Section 4 uses the spanning tree to find the convex hulls of points on either side of a query line, and then Section 5 applies this result to solve the original face-reporting problem on $\mathcal{A}(L)$. Section 6 discusses a general tradeoff between space and query time, Section 7 uses random sampling to obtain an efficient implicit representation for arrangements of segments, and Section 8 presents the application of calculating many faces at once.

2. Geometric Preliminaries

We make a few definitions before we begin. The convex hull of a set S of points is denoted by $h(S)$. Given a (nonvertical) line l, let l^+ be the closed halfspace above the line and let l^- be the closed halfspace below the line. We occasionally use l^* to refer to either l^+ or l^-.

In this paper we introduce "big-Λ" notation to suppress logarithmic and smaller factors. Specifically, if f and g are functions of n, we write $f(n)=\Lambda(g(n))$ if, for some constant k, we have $f(n)=O(g(n)\log^k n)$.

Let $L=\{l_1, l_2, \ldots, l_n\}$ be the set of n lines whose arrangement, $\mathcal{A}(L)$, we wish to store. We assume that no three lines of L pass through a common point. Given a query point p, the face containing p is bounded by the lines that p can "travel to" without crossing lines of L. We can map the set of lines L to a set of points $P=\{p_1, p_2, \ldots, p_n\}$ by a duality transform that preserves the above/below relationship between points and lines. The nondegeneracy of L implies that no three points of P are collinear. (See Edelsbrunner for a precise formulation of the duality [E].) A primal query point p becomes a dual nonvertical query line l. The lines of L bounding the face containing p now become points of P that the line l can "travel to" by translation and rotation without crossing any other point. In other words, the dual of the face consists of the portions of the convex hulls $h(P\cap l^+)$ and $h(P\cap l^-)$ between their inner common tangents, as depicted in Fig. 1. Thus our problem becomes one of finding quickly the convex hulls of the points of P lying either above or below a query line.

3. Computing Trees with Low Stabbing Number

In this section we define the stabbing number of a spanning tree on a configuration of points in the plane. We give three algorithms for computing trees with low

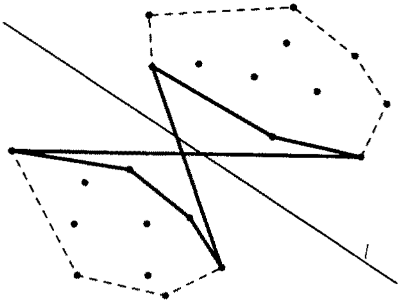

Fig. 1. The dual of a face.

stabbing number; these methods have tradeoffs between space, preprocessing time, stabbing number, and the use of randomization.

We begin with some definitions. Any line l that does not intersect P partitions the points of P into two sets—$l^+ \cap P$ and $l^- \cap P$. Two lines are *equivalent with respect to* P if they induce the same partition. We frequently use $\mathcal{L}$ to denote a set of lines such that every line that partitions P into two nonempty sets is equivalent to exactly one line of $\mathcal{L}$. We can establish a correspondence between the lines of $\mathcal{L}$ and pairs of points of P: If a line $l \in \mathcal{L}$ is rotated counterclockwise as far as possible without crossing points of P, it hits a unique pair of points, the left one from above and the right one from below. If the line determined by a pair of points of P is rotated slightly clockwise, so that it passes above the left point and below the right one, it is equivalent to a unique line of $\mathcal{L}$. Therefore there are $\binom{n}{2}$ lines in $\mathcal{L}$.

Let T be a (straight-edge) spanning tree on P. The *stabbing number of a line* l with respect to the tree T is the number of edges of T cut by l, that is, the number of edges whose endpoints come from different sets of the partition induced by l. The *stabbing number of* T is the maximum of the stabbing numbers of all lines with respect to T. We can prove the following lower bound:

Lemma 3.1. *There is a configuration of n points for which every spanning tree has stabbing number* $\Omega(\sqrt{n})$.

Proof. Let P be the configuration of size $n = m^2$ given by the points of an $m \times m$ grid. The arrangement of $2m - 2$ lines between the rows and columns of the grid isolates each point in a face; thus, each edge of the spanning tree must intersect one of these lines. The average number of edges intersected by a line is $(n-1)/(2m-2) \geq m/2 - 1$. So $\Omega(\sqrt{n})$ is a lower bound for the stabbing number of this configuration of n points. ☐

We have several methods for computing trees with stabbing number close to this lower bound. The first method gives a slow deterministic construction of a tree with optimal stabbing number $O(\sqrt{n})$; the second uses a conjugation partition tree, which can be built quickly, to make a spanning tree with stabbing number

$O(n^{0.695})$ [EW]. These methods are described in Section 3.1. The third method, described in Section 3.2, employs random sampling to find a tree such that, for any line l from the set of nonequivalent lines $\mathscr{L}$, the stabbing number of l is $\Lambda(\sqrt{n})$ with probability greater than one-half. We can then produce a family of $O(\log n)$ trees such that, with probability arbitrarily close to 1, any line l has stabbing number $\Lambda(\sqrt{n})$ with respect to one of the trees. Our method is based on drawing random samples from an implicitly specified set of lines; this is described in Section 3.3.

The first and third algorithms produce trees that may intersect themselves when embedded in the plane, so we also discuss how to untangle spanning trees without significantly increasing their stabbing number. In particular, in Section 3.4 we provide an algorithm that will, for the randomized construction, remove all self-intersections and at most double the stabbing number of each line.

3.1. *Preliminary Approaches*

This section describes two deterministic algorithms for building spanning trees with low stabbing number. Roughly speaking, these algorithms are at opposite ends of the tradeoff between stabbing number and preprocessing time and space. In Section 3.2 we show how introducing randomization lets us get close to the optimal stabbing number without incurring large preprocessing costs.

Theorem 3.2. *For a configuration of n points P, we can compute a spanning tree T with stabbing number* $O(\sqrt{n})$ *by a procedure that can be implemented to run in* $\Lambda(n^{3.5})$ *time and* $\Lambda(n^{1.5})$ *space.*

Proof. Chazelle and Welzl [CW] prove that the procedure descisced in the following two paragraphs computes a spanning tree with stabbing number $O(\sqrt{n})$.

Let T_1 be the graph with vertex set $P_1 = P$ and no edges. Let $\mathscr{L}$ be a set of lines such that every line that partitions P is equivalent to exactly one line in $\mathscr{L}$. Let $w_1\colon \mathscr{L} \to \mathbf{N}$ be a weight function that assigns integer 1 to each line in $\mathscr{L}$. For a pair of points $p, q \in P$ let $\mathscr{L}_{p,q} \subset \mathscr{L}$ be the set of lines that separate p and q.

Repeat the following steps for $i = 1, \ldots, n-1$: Find a pair $p, q \in P_i$ that minimizes the sum of the weights in $\mathscr{L}_{p,q}$. Add the edge (p, q) to T_i to obtain graph T_{i+1}. Remove q from P_i to obtain P_{i+1}. Define the weight function $w_{i+1}\colon \mathscr{L} \to \mathbf{N}$ as

$$w_{i+1}(l) = \begin{cases} 2w_i(l) & \text{if } \; l \in \mathscr{L}_{p,q}, \\ w_i(l) & \text{otherwise.} \end{cases}$$

After $n-1$ iterations, $T = T_n$ will be a spanning tree with stabbing number $O(\sqrt{n})$ (see [CW]).

This procedure is most naturally implemented by considering the dual problem. Each point $p \in P$ dualizes to a line l_p, edges (p, q) to double wedges $D_{p,q}$, and (the equivalence classes of) lines l in $\mathscr{L}$ to cells C_l in the arrangement of dual lines. The functions w_i then assign weights to cells of the arrangement and can be naturally extended to assign weights to any region R in the dual plane by $w_i(R) = \sum_{C \subset R} w_i(C)$.

The reader may convince himself that one iteration of the procedure outlined above in the primal corresponds to the following in the dual: Identify $p, q \in P_i$ so that $w_i(D_{p,q})$ is minimal. Add the edge (p, q) to T_i to obtain graph T_{i+1}. Remove q from P_i to obtain P_{i+1}. Define the weight function

$$w_{i+1}(C) = \begin{cases} 2w_i(C) & \text{if} \quad C \subset D_{p,q}, \\ w_i(C) & \text{otherwise.} \end{cases}$$

If the entire arrangement of dual lines is stored along with the weights for the cells, we can compute the weights of all double wedges by a fairly straightforward sweep algorithm. Unfortunately, the weights can become so large that it would be unreasonable to assume that arithmetic operations can be performed on them in time less than the number of bits necessary to represent them, which turns out to be $O(\sqrt{n})$ (see [CW]). As a consequence, one iteration takes time $\Lambda(n^{2.5})$, and the entire tree construction algorithm takes time $\Lambda(n^{3.5})$ and uses space $\Lambda(n^{2.5})$.

It is possible to reduce the space requirements to $\Lambda(n^{1.5})$ using a two-pass algorithm. Each pass sweeps a vertical line from left to right over the arrangement of dual lines, but does not represent the arrangement explicitly. The first pass computes the weight of the cells below each line. The second pass enumerates the $\binom{n}{2}$ intersection points of the dual lines in left-to-right order and computes the weights of the double wedges associated with them. Over the course of the sweep, the algorithm computes the double wedge of minimum weight.

To describe the algorithm, we introduce some notation. For each cell C, the weight $w_i(C)$ is equal to $2^{k_i(C)}$, where $k_i(C)$ is the number of edges (p, q) of T_i for which $C \subset D_{p,q}$. For a line l_p of the arrangement let $\overline{l_p}$ denote the lower halfplane bounded by l_p. For a vertical line l let l' denote the halfplane to the left of l.

The first phase of the algorithm sweeps a vertical line l from left to right over the arrangement while maintaining two data structures for the sweep line. One allows us to compute $k_i(C)$ whenever the sweep line reaches the rightmost vertex of a cell C. The other data structure maintains $w_i(\overline{l_p} \cap l')$ for all $p \in P_i$. We leave the details of these data structures to the reader and just remark that they can be implemented using space for $O(n)$ numbers so that updates and queries take $O(\log n)$ arithmetic operations. (Note that in the second data structure each arithmetic operation can take time $\Lambda(\sqrt{n})$ due to the size of the numbers involved.) At the completion of the sweep, $w_i(\overline{l_p})$ will have been computed for each $p \in P_i$; these values can be stored using $\Lambda(n^{1.5})$ space.

The second phase performs another sweep to determine the desired double wedge that has minimal weight. This sweep proceeds exactly as the first one does, using the same data structures. However, whenever the sweep line l reaches the intersection point of two lines l_p and l_q, with $p, q \in P_i$ and $slope(p) < slope(q)$, the weight of the double wedge $D_{p,q}$ can be computed by

$$w_i(D_{p,q}) = w_i(\overline{l_q}) - w_i(\overline{l_p}) + 2(w_i(\overline{l_p} \cap l') - w_i(\overline{l_q} \cap l')).$$

This works because our duality transform guarantees that each double wedge consists of left and right wedges, rather than top and bottom wedges. The weight of each double wedge is compared with the minimum weight seen so far.

During the sweeps $O(n^2)$ updates and queries are performed on the data structures, each at a maximal cost of $\Lambda(\sqrt{n})$ using space $\Lambda(n^{1.5})$. Thus the two sweeps take time $\Lambda(n^{2.5})$ and use space $\Lambda(n^{1.5})$, which yields a $\Lambda(n^{3.5})$ time, $\Lambda(n^{1.5})$ space algorithm for finding the desired spanning tree with low stabbing number. □

If we assume that addition and subtraction operations on integers with $O(\sqrt{n})$ bits take $O(\sqrt{n})$ time, then this sweepline algorithm runs in $O(n^{3.5} \log n)$ time. The running time can be improved to $O(n^{3.5})$ by using a topological sweep [EG], but we do not give details of that method here.

The spanning tree produced by the method of Theorem 3.2 may intersect itself. Section 3.4 tells how to transform the tree into one without self-intersections while at most doubling its stabbing number.

For our purpose, the time bound of the algorithm of Theorem 3.2 is excessive; given $O(n^2)$ time and space, we could explicitly construct the arrangement of lines and use it to answer queries. However, if we can accept a larger stabbing number, then we can apply a second deterministic technique to find a spanning tree from a partition tree using much less preprocessing time.

Theorem 3.3. *For a configuration of n points P, we can compute a non-self-intersecting spanning tree T with stabbing number $O(n^{0.695})$ by a procedure that uses $O(n \log n)$ time and linear space.*

Proof. Partition trees, such as Willard trees [Wi] or the conjugation trees of Edelsbrunner and Welzl [EW], recursively divide a set of points for efficient processing of halfplane range queries. Each node of the tree is associated with a subset of the points. The important property of partition trees is that, for any line l, at most $O(n^\alpha)$ nodes contain sets of points that are separated by l. The conjugation tree, for example, is a binary tree with $\alpha = \log_2((1+\sqrt{5})/2)$; it can be built deterministically in $O(n \log n)$ time.

We construct a graph with stabbing number $O(n^\alpha)$ from a conjugation tree by computing the convex hull of the set of points at each node. Using a divide-and-conquer strategy, we merge the convex hulls of the two children of a node by finding two outer common tangents. (In a conjugation tree, the children of a node are separable by a line; this guarantees that their convex hulls have exactly two outer common tangents.) Two hulls with m points can easily be merged in $O(m)$ time, which gives an $O(n \log n)$ bound for the whole process.

This divide-and-conquer procedure terminates with a planar embedding of a connected graph that has at most two edges for each node of the conjugation tree. Compute a spanning tree T on this graph. Any line l separates points in at most $O(n^\alpha)$ nodes of the conjugation tree, and thus T has stabbing number $O(n^\alpha) = O(n^{0.695})$. ☐

3.2. Randomized Algorithms

To obtain either fast preprocessing or optimal stabbing number, the deterministic algorithms of the preceding subsection sacrifice the other. If we use randomization, we can in $O(n \log n)$ time compute a tree T with an average stabbing number nearly as good as that of Theorem 3.2. More precisely:

Theorem 3.4. *Given a configuration of n points P, a random sample with size* $(1+\sqrt{2})\sqrt{n}$ *drawn from a set of lines* $\mathscr{L}$, *and a constant* $\alpha > 0$, *we can construct a spanning tree T on P such that, with probability* $1 - n^{-\alpha'}$ *for any* $\alpha' < \alpha$, *the total number of intersections between* $\mathscr{L}$ *and T is* $\Lambda(|\mathscr{L}|\sqrt{n})$, *where the constant of proportionality also includes* α *as a factor. Thus, the stabbing number of the lines of* $\mathscr{L}$ *is* $\Lambda(\sqrt{n})$ *on the average. The construction takes* $O(n \log n)$ *time.*

Proof. In order to minimize the total number of line-edge intersections as we build our tree, we employ random sampling techniques due to Clarkson [C], which are closely related to the ε-nets used by Haussler and Welzl [We], [HW]. The theory of ε-nets says that, in a range space of finite Vapnik-Chervonenkis dimension d, a random sample of size r drawn from a space of n elements is an ε-net with probability $1-\delta$, if the sample size r is at least $\max((4/\varepsilon)\log(2/\delta), (8d/\varepsilon)\log(8d/\varepsilon))$. In our case the ranges will be triangles or lines and our $\delta = n^{-\alpha}$. Thus we can compute a constant c such that, with probability $1-\delta$, the random sample is an ε-net for $\varepsilon = (c\alpha \log n)/r$.

Our algorithm constructs the tree T in at most $\log n$ phases. It uses part of the random sample of lines from $\mathscr{L}$ in each phase; the unused sample lines are stored in a list, which initially contains the whole random sample. The algorithm maintains a current set of points, which it initializes to P; each phase discards at least half of the current set. If a phase begins with j points—call it phase j—we do the following. Obtain a random sample R of the lines of $\mathscr{L}$ by removing the first $\lfloor\sqrt{j/2}\rfloor$ lines from the list. Construct the arrangement $\mathscr{A}(R)$ in time linear in j [EOS], [CGL]. Triangulate each of the faces of this arrangement to form a subdivision of the plane, which we denote $\mathscr{T}(R)$. A short calculation shows that the number of triangular faces in $\mathscr{T}(R)$ is at most $j/2$ for $j > 2$. Place the j points into the triangles that contain them by performing a point location on each point. After $O(j)$ initial preprocessing, each of these j point locations takes logarithmic time [EGS2], [Ki], [ST]. Finally, within each triangle, form a (non-self-intersecting) spanning tree on the points it contains, choose a root vertex, and discard all points except the root. This leaves at most $j/2$ points. (If $j = 2$, we do not bother to pick R, but just form a single-edge spanning tree on the two points.)

Each phase gets an independent random sample from the lines of $\mathscr{L}$, but for convenience we bundle them into the single sample specified in the statement of the lemma. The total size of the random sample we need from $\mathscr{L}$, summed over all phases, is at most $\sum_{i>0}\sqrt{n/2^i} = (1+\sqrt{2})\sqrt{n}$.

We must do three things: prove that the algorithm gives a spanning tree, show that this tree has the right average stabbing number, and determine the time complexity of the algorithm.

We stop the algorithm when $j = 1$, so the last phase finds a spanning tree. Phase j computes at most $j/2$ rooted spanning trees and, by induction, the remainder of the algorithm computes a spanning tree on their roots. Thus, the graph returned is connected. Since $n - 1$ edges are used altogether, the result is a spanning tree.

As mentioned above, there is a constant c_1 (which here includes the multiplicative factor of α) such that, with probability $1 - n^{-\alpha}$, our sample R is an ε-net of size $\varepsilon = c_1 \log n/\sqrt{j}$ for triangular ranges. This means that for any triangle τ in $\mathcal{T}(R)$, at most $c_1|\mathscr{L}| \log n/\sqrt{j}$ of the lines of the set $\mathscr{L}$ intersect τ. Therefore, each edge we add contributes at most this number of line-edge incidences to the total. Summing over all $\log n$ phases, the total number of edge-line incidences is, with probability $(1 - n^{-\alpha})^{\log n} \geq 1 - n^{-\alpha} \log n$, at most

$$\sum_{\substack{\text{for each edge}\\ \text{in each phase } j}} c_1|\mathscr{L}| \frac{\log n}{\sqrt{j}} \leq c_1|\mathscr{L}| \log n \sum_{1 \leq i \leq n} \frac{1}{\sqrt{i}}$$

$$= 2c_1|\mathscr{L}|\sqrt{n} \log n(1 + O(1/\sqrt{n})).$$

Thus, the average stabbing number over the lines of $\mathscr{L}$ is $\Lambda(\sqrt{n})$.

The time required by the j point locations at $O(\log n)$ time apiece dominates the time of phase j. Since the j's are bounded by a geometric series, the total time complexity is $O(n \log n)$. □

Note that the theorem allows the random sample to contain duplicates. In fact, the theorem holds even when $|\mathscr{L}|$ is smaller than $(1+\sqrt{2})\sqrt{n}$.

To compare this theorem to Theorem 3.2, let $\mathscr{L}$ be the set of $\binom{n}{2}$ nonequivalent lines that separate the points of P. We can obtain a random line of $\mathscr{L}$ by choosing a random pair of points and rotating slightly the line that they define (note that the rotation is only conceptual). Thus the average stabbing number is $\Lambda(\sqrt{n})$ and we can be assured that at least half of the lines have stabbing number at most twice the average.

Theorem 3.4 is useful for applications in which the average stabbing number is important, but the worst-case stabbing number is not. The calculation of many faces in an arrangement of lines, described in Section 8, is such an application.

We use an iterative approach to obtain a good worst-case stabbing number. The algorithm of Theorem 3.4, running on a set of lines $\mathscr{L}$, produces a tree T with the property that at least half of the lines of $\mathscr{L}$ are *good*: they have stabbing number $\Lambda(\sqrt{n})$. If we could identify which lines were *bad*, then we could run the algorithm again on these lines. Repeating this process a logarithmic number of times, we would form a family of trees such that every line has stabbing number $\Lambda(\sqrt{n})$ with respect to one of the trees. To identify the bad lines explicitly seems expensive; however, by one more application of random sampling, we can approximate the set of bad lines closely enough to establish the following theorem.

Theorem 3.5. *Given a configuration of n points P and a constant* $\alpha > 0$, *we can compute a family of* $k = O(\log n)$ *spanning trees* $T_1, T_2, \ldots, T_k$ *on P with the property that, with probability* $1 - n^{-\alpha'}$ *for any* $\alpha' < \alpha$, *for every line l there is a tree* T_i *such that l has stabbing number* $\Lambda(\sqrt{n})$ *with respect to* T_i. *The procedure uses* $\Lambda(n^{3/2})$ *time and linear space for each tree in the family, for a total of* $\Lambda(n^{3/2})$ *time and* $O(n \log n)$ *space. Again,* α *enters as a multiplicative factor into the time complexity bound.*

Proof. To aid our presentation, we number the bad sets and the spanning trees according to the order in which they are produced. Tree T_i is built based on the bad set $\mathscr{L}_i$, and bad set $\mathscr{L}_{i+1}$ is defined by $\mathscr{L}_i$ and T_i. The first set, $\mathscr{L}_1 = \mathscr{L}$, is the set of $\binom{n}{2}$ nonequivalent lines that separate P in all possible ways. The first spanning tree, T_1, is good for at least $|\mathscr{L}_1|/2$ lines. We would like $\mathscr{L}_2$ to include the bad lines and none of the good lines; more generally, we would like $\mathscr{L}_{i+1}$ to include the bad lines of $\mathscr{L}_i$ and none of the good lines. However, since we do not allow ourselves enough time to count all intersections between $\mathscr{L}_i$ and T_i, this seems to be asking too much.

We resolve the problem by letting the set $\mathscr{L}_i$ approximate the set of bad lines. There are two challenges to be met: First, we cannot let bad lines slip out of the approximation, yet we need to reduce the size of $\mathscr{L}_i$ by a constant factor at each step. Second, we cannot store $\mathscr{L}_i$ explicitly—it is too large. We need to store it in an implicit form that still allows us to choose a random sample of $(1+\sqrt{2})\sqrt{n}$ lines quickly. It is the cost of sampling from $\mathscr{L}_i$ that dominates the cost of constructing the spanning trees: by Theorem 3.4, each spanning tree construction takes $O(n \log n)$ time, given a random sample from $\mathscr{L}_i$ of size $O(\sqrt{n})$, but, as Theorem 3.8 below shows, finding the random sample takes $\Lambda(n^{3/2})$ time.

Our method for computing the set $\mathscr{L}_i$ uses the fact that a random sample of the edges of T_i is a good estimator for the number of intersections of a line with T_i: if we take an appropriately sized random sample of tree edges, we expect bad lines to reveal themselves by intersecting the sample.

We pick a random sample R of $r = c_2\sqrt{n}/\log n$ tree edges, where the constant c_2 is chosen so that at most $|\mathscr{L}_i|/2$ lines intersect the sample. The lines of $\mathscr{L}_i$ that intersect an edge of the sample constitute the new bad set approximation, $\mathscr{L}_{i+1}$. In the next two paragraphs, we show how to choose the constant c_2 to obtain a sample with at most $|\mathscr{L}_i|/2$ points of intersection between $\mathscr{L}_i$ and R—this will certainly ensure that $|\mathscr{L}_{i+1}| \le |\mathscr{L}_i|/2$. Then we show that the lines that do not intersect an edge of R stab few edges of T_i.

How do we bound the number of intersections between $\mathscr{L}_i$ and our random sample R? The construction of T_i gives a bound on the number of intersections between each edge of the tree and the lines of $\mathscr{L}_i$. In fact, the construction of T_i gives a bound on the number of lines passing through the triangle containing edge e when e is added to the tree; the algorithm can store that bound as $B(e)$. Since the average bound on the number of line intersections with an edge of T_i is $2c_1|\mathcal{L}_i| \log n/\sqrt{n}$, the expected number of intersections between $\mathcal{L}_i$ and R (the random sample of $r = c_2\sqrt{n}/\log n$ edges of T_i) is bounded by $2c_1c_2|\mathcal{L}_i|$. Choose c_2 so that $2c_1c_2 = \frac{1}{4}$. Then at least half of all the random samples of size r have at most $|\mathcal{L}_i|/2$ intersections with $\mathcal{L}_i$, according to the bounds $B(e)$.

For a particular sample R, we sum the intersection bounds of its edges, $\sum_{e\in R} B(e)$, in order to bound the number of intersections between R and $\mathcal{L}_i$. If the bound is greater than $|\mathcal{L}_i|/2$, we pick another sample. Since summing the bounds takes only $O(\sqrt{n})$ time, in linear time we can choose up to $\sqrt{n}$ samples if necessary. Thus, with probability $1-2^{-\sqrt{n}}$, we can guarantee that the sum of the bounds is at most $|\mathcal{L}_i|/2$. If the construction of tree T_i succeeded, then the bounds are accurate, and $|\mathcal{L}_{i+1}| \leq |\mathcal{L}_i|/2$.

We use ε-nets to bound the stabbing number of lines of $\mathcal{L}_i - \mathcal{L}_{i+1}$. The elements of the range space are tree edges and the ranges are lines. We again choose δ to be $n^{-\alpha}$; this implies that $\varepsilon = \Theta(\log n/r)$. In this application of random sampling, we reject as many as half of the samples, so the proportion of times the sample fails to be an ε-net may as much as double. But we can still compute a constant c' such that with probability at least $1-2n^{-\alpha}$, any line that cuts at least $c'\sqrt{n}\log^2 n$ edges of T_i also cuts a line of the random sample. Therefore, each line deleted from $\mathcal{L}_i$ has stabbing number $\Lambda(\sqrt{n})$ with respect to tree T_i.

What is the probability that all the bounds on stabbing numbers are correct, and hence that the algorithm succeeds? With probability $1-2^{-\sqrt{n}}$, we can find a random sample R that has a low intersection bound with $\mathcal{L}_i$. Independently, each sample has the ε-net property with probability $1-2n^{-\alpha}$, and each spanning tree T_i has the right stabbing bounds with probability $(1-n^{-\alpha})^{\log n}$. Thus the overall probability of success is at least

$$(1-2n^{-\alpha}-2^{-\sqrt{n}})^{2\log n}(1-n^{-\alpha})^{2\log^2 n} \geq 1-(4\log n+2\log^2 n+1)n^{-\alpha} \geq 1-n^{-\alpha'}$$

for any $\alpha' < \alpha$ as n goes to infinity.

Let us summarize our algorithm. In a sequence of $k = O(\log n)$ stages, the algorithm produces k spanning trees for P. Stage i draws a random sample of size $(1+\sqrt{2})\sqrt{n}$ from the approximate bad set $\mathcal{L}_i$. We show in Theorem 3.8 below that drawing the sample from $\mathcal{L}_i$ takes $O(n^{3/2}\log n)$ time and linear space for all stages after the first (sampling from $\mathcal{L}_1 = \mathcal{L}$ takes $O(\sqrt{n})$ time). The algorithm uses the random sample from $\mathcal{L}_i$ to compute T_i in $O(n \log n)$ time by the method of Theorem 3.4. It then draws a random sample of the edges of T_i and defines $\mathcal{L}_{i+1}$ implicitly as the set of lines in $\mathcal{L}_i$ that intersect these edges. To bound the time and space required by the algorithm, we impose a mixed termination condition: the algorithm terminates when $\mathcal{L}_{i+1}$ is empty or after $2\log n$ stages, whichever comes first. With high probability, $\mathcal{L}_{i+1}$ contains at most half the lines of $\mathcal{L}_i$ at each stage, so the algorithm almost always terminates with $\mathcal{L}_{i+1}$ empty. Let k be the number of trees built by the algorithm. The space needed to store the trees is $O(kn) = O(n \log n)$, and the time used to build them is $O(n^{3/2}\log^2 n)$. The properties of random samples ensure that, with probability at least $1-n^{-\alpha'}$

for any $\alpha' < \alpha$, every line l has stabbing number $O(\sqrt{n} \log^2 n)$ with respect to at least one of $T_1, T_2, \ldots, T_k$. ☐

3.3. *Random Sampling from a Bad Set*

Theorem 3.5 defines the approximate bad set $\mathscr{L}_{i+1}$ as the set of all lines in $\mathscr{L}_i$ that intersect a random sample of the edges of T_i. This subsection shows how to sample from $\mathscr{L}_{i+1}$ without looking at all of its lines. We begin by considering the definition of the lines in $\mathscr{L}_1 = \mathscr{L}$.

Each line l of $\mathscr{L}$ is determined by two points of P: we draw l through the two points and rotate it slightly clockwise, so that it passes above the left point and below the right one. But recall that the points of P are really the duals of the original n lines of L. In primal space, then, l corresponds to an intersection of two original lines, perturbed slightly into one of the four quandrants defined by the lines (the quadrant above the line whose dual is the left point, and below the line whose dual is the right point). We refer to the primal version of l as the *canonical perturbation* of its corresponding intersection. Line l cuts a (dual) segment if l's corresponding primal point lies in a double wedge corresponding to the segment. (The bounding lines of the double wedge are the primal versions of the segment endpoints.) Define W_i to be the (primal) arrangement of the double wedges that correspond to all the randomly sampled edges from $T_1, \ldots, T_{i-1}$. Each face of W_i dualizes to the set of all lines that cut a particular subset of the sample edges. A line is in $\mathscr{L}_i$ if and only if it cuts a sampled edge from each tree $T_1, \ldots, T_{i-1}$; equivalently, a line belongs to $\mathscr{L}_i$ if and only if its primal version falls in one of a certain set of faces of W_i. We think of these faces as being colored. Translating the definition of $\mathscr{L}_{i+1}$ in Theorem 3.5 to primal terms, we get the colored faces of W_{i+1} by intersecting the colored faces of W_i with the set of double wedges that correspond to the random sample from T_i. Since W_i has complexity $O(ni^2/\log^2 n) = O(n)$, we can build it and determine its colored faces in linear time [CGL], [EOS].

The arrangement W_i defines $\mathscr{L}_i$ implicitly. To build T_i based on $\mathscr{L}_i$, we need to draw random samples from $\mathscr{L}_i$. In primal terms, we want to pick a random sample from the intersections of the lines of L (the original n lines) that fall in colored faces of W_i, or whose canonical perturbations do so. We refer to these intersections, which fall in colored faces, as being colored themselves. To enumerate all the intersections of primal lines explicitly would take $O(n^2)$ time; we enumerate them implicitly using $\Lambda(n^{3/2})$ time and $O(n)$ space. Our implicit enumeration is based on the fact that there are only $O(n^{3/2} i / \log n)$ intersections between the n original lines of L and the edges of W_i. We compute these intersections, then use them to count and sample from the colored intersections.

The idea of the sampling procedure is the following: We count the colored intersections, implicitly assigning a unique integer index to each. We then pick a random sample from the integers between 1 and the number of colored intersections. Our counting algorithm lets us explicitly produce the intersection corresponding to any given index, so we run the algorithm again to produce the intersections that correspond to the randomly chosen integers. These intersections constitute the desired random sample.

The algorithm for counting and computing intersections has two distinct parts, one dealing with a single face of W_i and one dealing with all the faces. The single-face algorithm takes as input the set of all lines of L that intersect the boundary of a given colored face. It counts the intersections of lines of L that fall inside the face and implicitly assigns indices to them. Given a sorted set of indices, it can produce the intersections corresponding to them. The global part of the algorithm provides the input to the single-face algorithm. It enumerates the colored faces of W_i and computes the intersections of L with the boundary of each face. Both parts of the algorithm use $O(\log n)$ time per intersection between L and W_i and linear space altogether. The following lemma describes the single-face algorithm.

Lemma 3.6. *Let f be a colored face of W_i, and let H be the set of lines of L that intersect the boundary of f. Define $h = |H|$. If we are given H as input, then in $O(h \log h)$ time and $O(h)$ space we can count the intersections of L inside f, including those on the boundary whose canonical perturbations lie in f. Let the number of these intersections be I. The counting algorithm implicitly assigns a unique integer in the range 1 to I to each intersection; given a sorted list of integers between 1 and I, the algorithm can produce the corresponding intersections in $O(\log n)$ additional time per intersection during the counting process.*

Proof. We restrict our attention to the lines in H, since any line of L that intersects the interior of face f must cross the boundary of f, and therefore must belong to H. Our algorithm has two steps, one to count intersections on the boundary of f and one to count intersections inside f.

The first step of the algorithm sorts the intersections of H with the boundary of the face f in order around f. This takes $O(h \log h)$ time. (Because H includes the supporting lines of f, our analysis does not need to include the complexity of the boundary of f explicitly.) From the sorted list the algorithm determines the $O(h)$ intersections of lines of L that fall on the boundary of f. It counts explicitly the ones whose canonical perturbations fall inside the face f.

The second step of the algorithm counts the intersections of lines in the interior of f. It considers only the transverse intersections between the lines of H and the boundary of f, those whose lines cut the interior of f. It numbers these intersections in order around f from 1 to h', for some $h' \leq 2h$. The intersections are paired: each intersection stores the index of the other intersection of its line with the boundary of f. The algorithm to count the intersections inside the face is similar to counting inversions in a permutation [Kn]. The algorithm maintains a list S of active intersections, which it initializes to empty, and a count *count* of intersections noted so far. The counting algorithm scans through the intersection indices in order from 1 to h'. At each step, let a be the current index. The line that intersects the face f at a cuts it a second time, at a point with index b. If $a < b$, then neither is in S; we append a to S. On the other hand, if $a > b$, then b already belongs to S; we locate b in S, set c to be the number of elements of

S after b, add c to *count*, and delete b from S. The same technique has been used in [GOS] to count intersections between line segments.

The correctness of the second step of the counting algorithm rests on the observation that the quantity c is the number of lines that intersect segment ab and have not yet had their intersections counted. The list S is a sorted list of integers that supports the operations *append*, *find*, *find-by-index* (given j, return the jth element of S), *delete*, and *length-of-tail* (return the number of elements after the given one). Each operation takes $O(\log h)$ time if S is represented by a balanced binary tree. The algorithm performs $O(h)$ operations, each of which takes $O(\log h)$ time; this gives an $O(h \log h)$ overall time bound. The space requirement is $O(h)$.

Given a sorted list of indices, the algorithm can produce the corresponding intersections in order as it runs. The intersections on the boundary of f are trivial to produce. Consider an index x belonging to an intersection in the interior of f. The algorithm runs as described above until $count < x \leq count + c$. Then the desired intersection is the one involving ab and the line whose intersection with the face boundary is distance $(x - count)$ after b in S. The intersection is computed using *find-by-index*. □

We have seen how to count the intersections of the n original lines of L that fall inside a face of W_i, given the intersections of L with the boundary of the face. We must now show how to enumerate the colored faces of W_i in some order, and for each face find all intersections of L with its boundary. Our goal is to do the enumeration in $O(n)$ space and in $O(\log n)$ time per intersection between L and W_i. We achieve these bounds by enumerating the faces along each line of W_i successively (each double wedge contributes two lines to W_i).

In outline, our method is the following: We associate each colored face of W_i with the unique supporting line below it of minimum slope. Our duality transform ensures that this line exists, since no double wedge can contain the bottom face of W_i (see the proof of Theorem 3.2). Next we step through the lines of W_i in some arbitrary order. For each line $l \in W_i$, we find the intersections of L with the faces associated with l, then count the intersections in each such face. We process one face at a time, taking them in left-to-right order. The algorithm uses the following ordering lemma:

Lemma 3.7. *Let $f_1, f_2, \ldots$ be the faces of an arrangement that are incident to a line l of the arrangement and lie on one side of l (say above l). Then any other line l' that intersects some of $f_1, f_2, \ldots$ intersects them in a left-to-right order consistent with their order along l, or the reverse of that order.*

Proof. If the lemma is violated, then there are faces a, b, and c that appear along l in order a, b, c (either left-to-right or right-to-left order) and along l' in order a, c, b. We assume without loss of generality that the order along l is left to right. Consider the line l_c of the arrangement whose intersection with l creates the left vertex of c on l. See Fig. 2. Now a and b lie on the opposite side of l_c

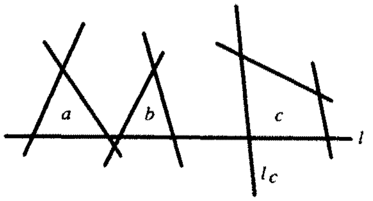

Fig. 2. Any line intersects faces a, b, and c in order.

from c, as do all predecessors of c along l. For l' to intersect a, c, and b in that order, it must cross l_c twice, which is impossible. □

This lemma lets us produce the intersections of a line l' with the faces associated with l in left-to-right order (with respect to l). We can use ray-shooting [CG1] or fractional cascading [CG2] to compute which of these faces l' hits in order along l'; by the lemma, this order is consistent with the order of the faces along l. We use a *next-face* function that, given l' and its current face, in $O(\log n)$ time computes the next face associated with l that l' hits.

Given a line l of W_i, our method begins by scanning through the colored faces incident to l and determining which of them are associated with l; we speak of these faces as being *flagged* in the remainder of this paragraph. The scan takes $O(i\sqrt{n}/\log n)$ time [CGL], [EOS]. Each flagged face has a bucket that holds lines of L; each line of L goes in at most one bucket. We maintain the invariant that when a flagged face is processed, its bucket holds all the lines of L that intersect the face. We initialize the buckets by using *next-face* to place each line of L in the bucket for the leftmost flagged face (with respect to l) that it intersects. We next scan through the flagged faces along l from left to right. When we reach face f, we count the intersections of L inside f as described in Lemma 3.6. When we finish with face f, we apply *next-face* to each line l' of L that hits f, obtaining the next intersection of l' with a flagged face. We transfer l' from the bucket for f into the bucket for the next intersected face. This guarantees that when each face is processed, its bucket contains all the lines of L that intersect it, as required.

This method lets us count the intersections of L that fall in colored faces of W_i. How do we use this to take a random sample from the colored intersections? Once we know how many colored intersections there are altogether, we generate a random sample of $(1+\sqrt{2})\sqrt{n}$ integers between 1 and the number of colored intersections, then run the algorithm again, this time using it to produce the intersections indexed by the random numbers. This produces a random sample of the colored intersections.

What are the time and space complexities of this algorithm? For each line of W_i, the algorithm uses $O(n \log n)$ time to initialize the buckets. It takes $O(\log n)$ time to find each face-line intersection. The face processing costs also amount to $O(\log n)$ per intersection. Therefore, since there are $O(n^{3/2}i/\log n)$ intersections between L and W_i, selecting a random sample from $\mathscr{L}_i$ takes $O(n^{3/2}i)$ time, or a total of $O(n^{3/2}\log^2 n)$ for all $1 \le i \le 2\log n$. The space complexity is linear: $O(n)$ for the arrangement W_i, plus an additional $O(n)$ for the buckets.

We have established the following theorem, which was required in the proof of Theorem 3.5:

Theorem 3.8. *Given the set of spanning tree edges from trees* $T_1, \dots, T_{i-1}$ *that determines the set of lines* $\mathscr{L}_i$, *we can define the arrangement* W_i *of the double wedges whose duals are these tree edges. We define some faces of* W_i *to be colored in accordance with the definition of* $\mathscr{L}_i$: *a line of* $\mathscr{L}$ *belongs to* $\mathscr{L}_i$ *if and only if it is the dual of a point lying in a colored face of* W_i. *We can build* W_i *in* $O(n)$ *time and space. Using* W_i, *we can take a random sample of size r from* $\mathscr{L}_i$ *in* $O((n^{3/2}+r)\log n)$ *time and* $O(n+r)$ *space.*

3.4. Untangling a Spanning Tree

The algorithms of Theorems 3.2 and 3.4 can generate trees that have self-intersections—we call such trees *tangled.* In order to use these trees in our application, we must untangle them without increasing their stabbing numbers by too much. We give an algorithm that takes the output of the algorithm of Theorem 3.4 and *untangles* it—returning a tree without self-intersections—in $O(n \log n)$ time. The algorithm can be adapted to untangle a tree from Theorem 3.2 in $\Lambda(n^{3/2})$ time. The algorithm has two parts. First, we form a tree with no self-intersections that has at most n additional vertices, or *Steiner points.* Second, we eliminate the Steiner points by employing a result of Guibas and Hershberger [GH] to find shortest paths from vertex to vertex using the tree as a simple polygonal obstacle. The first part does not increase the stabbing number of any line; the second at worst doubles the stabbing number of any line.

At times during the algorithm, we want to think of an untangled tree as a simple polygon. We can do so by tracing a Eulerian tour around the tree and treating it as a simple polygon with twice as many edges. As a result, many basic algorithms designed for simple polygons apply to our trees as well. For example, Chazelle and Guibas [CG1] give an algorithm that can be adapted to our structure so that, with $O(n \log n)$ preprocessing and $O(n)$ space, we can answer "shooting queries" in $O(\log n)$ time. A shooting query has the following form: given a ray from a point p, report the intersection point closest to p of the ray with an untangled tree. This will be an important tool in the proof of Theorem 3.9.

Theorem 3.9. *Suppose we are given the edges of a tangled spanning tree on n points that are output by the algorithm of Theorem* 3.4. *Then we can construct an untangled tree with at most n additional* (*Steiner*) *points by using* $O(n \log n)$ *time and linear space. For any line, the stabbing number with respect to the untangled tree will be at most the stabbing number with respect to the tangled tree.*

Proof. Recall that Theorem 3.4 constructs a spanning tree in phases, so that the new edges generated within each phase do not intersect one another. We build the desired *Steiner tree* (a non-self-intersecting tree with at most n additional vertices, or Steiner points) in an incremental manner, adding the edges one at a time in a way that makes the union of all added edges connected at all times. That is, we now add edges in the order opposite to the one in which the edges were found. We say that the untangling algorithm runs in *stages* to distinguish them from the *phases* of the tree-construction algorithm. We call the points of the configuration P *real* points or *real* vertices.

In the beginning of stage j, we assume that we have a Steiner tree with at most j (real and Steiner) vertices and without self-intersections. We can preprocess the Steiner tree for shooting queries as mentioned above. We also have several trees made up of the edges that were found in phase j of the tree-construction algorithm—since each one is rooted on the Steiner tree, we call them *rooted trees.* Taking the Steiner tree along with all the rooted trees gives a spanning tree of all j points. Furthermore, no two edges from rooted trees intersect.

Within each rooted tree, number the vertices by a preorder traversal from the root; a child receives a higher number than its parent. Order the edges by labeling them with the greater of their two endpoint numbers. If we respect this order as we add edges to the Steiner tree, then the path from a vertex p to the root is added before any edge into p from one of its children.

Suppose we want to add the edge (c, p), where vertex c has a larger number than vertex p. Shoot from the child c toward the parent p; the shot can hit either p or an edge of the Steiner tree. In the former case, we add edge (c, p) to the tree. In the latter, we introduce a Steiner point s at the first intersection point from c toward p, and add edge (c, s). Point s splits a Steiner tree edge into two edges.

This process maintains connectivity and clearly introduces no cycles, since it always adds as many vertices as edges.

At the end of stage j, we have performed shooting from a total of $j-1$ vertices and so have introduced at most $j-1$ Steiner points. We have a Steiner tree with no self-intersections and fewer than $2j$ vertices. The number of vertices will at least double at the next stage. Thus, we satisfy the initial assumption of the next stage and can proceed to it.

After stage n, we have a Steiner tree with fewer than n Steiner vertices. Since the shooting process merely shortens the original edges, the stabbing number of a line with respect to the tree cannot increase. The time taken is bounded by a geometric sum:

$$O\left(\sum_{\substack{\text{for each}\\ \text{stage } j}} j \log j\right) = O\left(\left(n + \frac{n}{2} + \cdots + 1\right) \log n\right) = O(n \log n).$$

This completes the algorithm. □

We can use the same approach to build a Steiner tree from the output of Theorem 3.2, or, in fact, from any tangled tree. We traverse the tangled tree in preorder. When we visit a vertex v, we shoot along the edge from v toward its parent, then add to the Steiner tree the edge from v to the first intersection of the shot with the current Steiner tree. Determining the first intersection is slow, in general, but in the special case of a tree with low stabbing number, we can determine all intersections of tree edges, and hence the ones we need, in $\Lambda(ns)$ time, where s is the stabbing number of the tree [BO], [CE].

In our application, additional Steiner vertices in the tree correspond to additional lines in the primal arrangement. These lines could hide true edges or introduce dummy edges in the faces of the arrangement and thus affect the answers to queries. To remove the Steiner points, we use the shortest path algorithm of Guibas and Hershberger [GH] as follows: Consider an Eulerian tour τ around the outside of the tree; it visits both Steiner points and real vertices and has stabbing number at most double that of the Steiner tree. For any pair of real vertices p and q such that the portion of the tour τ between them visits only Steiner points, we replace this portion by the shortest path from p to q that does not intersect the Steiner tree. We show that the resulting straight-edge multigraph G is connected, has only real points for vertices, and has at most $O(n)$ edges. The edges of G do not cross each other. Moreover, the stabbing number of G with respect to any line is at most that of τ.

Two real vertices p and q are said to be *adjacent* on the tour if the tour contains a path from p to q that uses only Steiner points. We call the path a *Steiner path* and denote it by $Stp(p, q)$. Let $sp(p, q)$ denote the Euclidean shortest path from p to q that avoids the tree. In our algorithm, we replace the Steiner path $Stp(p, q)$ between adjacent vertices p and q by the shortest path $sp(p, q)$. This may turn our tour into a graph with higher degree vertices, but it maintains connectivity and, as we show, it does not increase the stabbing number of any line. We have one important observation about Steiner points.

Lemma 3.10. *With respect to the Eulerian tour, all Steiner points are convex vertices.*

Proof. Steiner points are introduced along an edge, forming one angle of 180° and two angles that sum to 180°. □

We call a simple polygon a *spiral from p to q* if the two polygonal chains from p to q both turn only to the left or only to the right.

Lemma 3.11. *Between adjacent real vertices p and q, the Steiner path and shortest path, $Stp(p, q)$ and $sp(p, q)$, form a spiral from p to q. This spiral contains no vertices of the Steiner tree in its interior.*

Proof. By construction, the Steiner path goes along the tree and the shortest path never touches the Steiner path, as shown in Fig. 3. Thus, concatenating the Steiner and shortest paths gives a simple polygon, S.

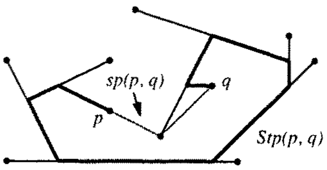

Fig. 3. The Steiner and shortest paths form a spiral.

If we take the convex hull of our tree, all the hull vertices are real vertices. Thus we will never have a Steiner path that leaves one bay or pocket of the hull and goes to another. The shortest path $sp(p, q)$ lies in the same bay as $Stp(p, q)$. Because the polygon S does not cross the Steiner tree, it must contain either all or none of it; because S lies in a single bay of the convex hull, it therefore contains no vertex of the Steiner tree.

Lemma 3.10 implies that there are no reflex vertices on the Steiner path, so it turns in only one direction. If the path $sp(p, q)$ made a convex turn with respect to the inside of S, then the path could be shortened. Thus the shortest path turns in only one direction—the same direction as the Steiner path. □

Since the shortest paths have reflex angles with respect to the tour, they use no Steiner points. Thus the graph of all shortest paths between adjacent real vertices uses only real vertices.

We can now prove that the stabbing number of the shortest path $sp(p, q)$ is less than or equal to that of the Steiner path $Stp(p, q)$. In a spiral, the chain with reflex angles on the inside of the spiral is called the *inner* chain, and the other chain is the *outer* chain. For example, shortest path $sp(p, q)$ is the inner chain of the spiral from p to q. If p and q are both on the convex hull of the spiral from p to q, then we call it a *short spiral.*

Lemma 3.12. *Let S be a spiral from p to q. Any line intersects the outer chain at least as often as the inner chain.*

Proof. If the spiral is short, then the conclusion holds. Thus we will cut a long spiral into several short spirals by the following process. Consider walking along the inner chain from p to q, maintaining a point w on the inner chain and a ray that is tangent to the inner chain at w and opposite the direction of the walk. As we walk on the inner chain from p, this ray sweeps a point w' on the outer chain.

At the beginning of the walk, choose a direction d—perhaps the direction of the first edge on the walk. Whenever our walk faces direction d or $-d$, we cut the spiral along the edge (w, w'). See Fig. 4. Since w' is a convex angle, we form a short spiral from p to w that includes edge (w, w') on the outer chain. Since the cut is tangent to the inner chain at w, we have a spiral from w' to q with edge (w, w') on the inner chain. We continue walking and cutting until only short spirals remain.

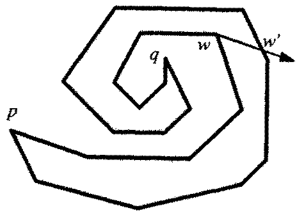

Fig. 4. Cutting a long spiral along a ray.

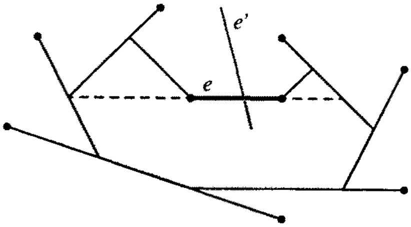

Fig. 5. No two shortest path edges cross.

Any line that intersects an added edge does so once as an inner and once as an outer edge. Thus, the added edges do not contribute to the difference between the number of intersections of a line with the inner and outer chains. Since the conclusion holds for each of the short spirals, it holds for the original spiral. □

Before we show how to find the graph of adjacent shortest paths, we bound its size. We prove the following lemma.

Lemma 3.13. *Let G be the multigraph obtained by replacing each Steiner path $Stp(p, q)$ between a pair of adjacent real vertices, p and q, with the shortest path avoiding the tree, $sp(p, q)$. The number of (straight) edges in G is $O(n)$.*

Proof. We first show that G is planar. Let e be any edge on the inner chain of a spiral. If we extend e in both directions until it hits the spiral boundary, it will hit the outer chain of the spiral (because all vertices on the inner chain are reflex). See Fig. 5. Any edge e' that crosses e without crossing the outer chain must end in the shaded region of Fig. 5; in particular, since the only real vertices on the shaded region are the endpoints of e, at least one endpoint of e' is not a real vertex. Because the edges of a shortest path end at real vertices, no shortest path edge can cross e. Thus no two edges of G can cross: G is planar.

Next we must show that any one edge is not used in too many shortest paths. Let e be an oriented edge of G. Edge e belongs to some spiral, which lies either to the left or to the right of e. Because the interior of a spiral is empty, if we extend the perpendicular bisector of e into the spiral, we must hit the outer chain of the spiral. This first intersection of the bisector with the Steiner tree is unique, and it lies on a unique Steiner path, whose endpoints therefore define a unique spiral. Thus at most two spirals contain e, one to its left and one to its right.

We have shown that G is a straight-edge planar multigraph on n vertices in which each edge appears at most twice. Therefore, G has $O(n)$ edges. □

We now have enough machinery to indicate how Guibas and Hershberger's shortest-path algorithm constructs the graph of shortest paths between adjacent vertices, and to analyze the amount of time and space required.

Theorem 3.14. *Given a Steiner tree, in $O(n \log n)$ time and $O(n)$ space we can construct a spanning tree of the original points that has no Steiner points, and whose stabbing number with respect to any line is at most twice that of the Steiner tree.*

Proof. Take an Eulerian tour τ around the Steiner tree—doubling the stabbing number of each line—and identify the pairs of adjacent real vertices on the tour. Using the algorithm of Guibas and Hershberger [GH], we can find the shortest path between any pair of points p and q in time $O(\log n+k)$, where k is number of edges along the path.

As a preprocessing step for finding shortest paths, triangulate the bays defined by the Steiner tree and its convex hull in $O(n \log n)$ time. The algorithm of Guibas and Hershberger [GH] requires a linear amount of preprocessing once a triangulation is given. It finds the shortest path $sp(p, q)$ in time $O(\log n+k)$, where k is the number of edges on $sp(p, q)$. In our application, Lemma 3.13 proves that the total number of edges on all shortest paths is $O(n)$; thus the logarithmic overhead for each path determines the running time, giving $O(n \log n)$ in total.

Lemmas 3.11 and 3.12 prove that, for any line, the stabbing number of the graph of all shortest paths is no larger than that of the Eulerian tour. Building a spanning tree on this graph completes the algorithm. □

4. Representing and Computing Convex Hulls

This section takes the $O(\log n)$ spanning trees provided by Section 3 and transforms them into a data structure that supports face queries on $\mathscr{A}(L)$. We describe a query algorithm that, given a query point p, returns an implicit representation of the face of $\mathscr{A}(L)$ that contains p. If the dual line of p has $\hat{s}$ as its minimum stabbing number with respect to any of the spanning trees, then the query takes $\Lambda(\hat{s})$ time. The implicit representation returned by the query can be used to find any of several face properties (intersections with a line, extreme vertices, the edge below p, and the number of face edges, for example) in additional $O(\log n)$ time, or to list the K face edges in additional $O(K+\log n)$ time.

Our algorithm uses geometric duality to answer face queries. The dual of the face of $\mathscr{A}(L)$ that contains p is an "hourglass" shape in the configuration P, as shown in Fig. 1. The "hourglass" is defined by the inner common tangents of the two convex hulls $h(P \cap l^+)$ and $h(P \cap l^-)$. Our algorithm computes the two hulls, finds the common tangents, and transforms the results to answer the primal query. Its basic operation is computing $h(P \cap l^*)$, for $l^* \in \{l^+, l^-\}$.

We transform the trees of Section 3 into simple paths, which are easier to work with than trees. For a particular tree, we trace around the boundary of the tree to get an Eulerian tour, then drop an edge to get a path. Every nonleaf vertex of the tree appears more than once along the path. However, in what follows we assume that the path is simple—that overlaps do not occur. We can simulate this condition during the running of our algorithm using standard techniques. Let Γ denote the simple polygonal path that we get from the spanning tree.

We first preprocess Γ to get a structure called a *bridge tree* that represents the convex hull of a simple polygonal path and supports several operations on that hull, including finding tangents. Let the s subpaths of Γ included in the query halfplane l^* be $\gamma_1, \ldots, \gamma_s$. We determine the γ_i's by intersecting l with Γ in

$O(s \log n)$ time [CG2]. (Actually, if we use the algorithm of Theorem 3.5, which builds $O(\log n)$ trees, we have $O(\log n)$ paths, and we want to use the one with fewest intersections with l. To find it, we run the intersection-finding algorithm of [CG2] in lock-step on all the paths at the same time and stop when all intersections with one of the paths have been found. This path is denoted Γ. This takes $O(s \log^2 n)$ time altogether.) We compute the bridge tree for each γ_i, then use these bridge trees to find $h(\gamma_1 \cup \cdots \cup \gamma_s)$.

We now give an intuitive description of bridge trees, then formally list their properties. Implementation details are omitted here, but will appear in a forthcoming paper on bridge trees [GHS]. A bridge tree $\mathscr{B}(\gamma)$ represents the convex hull of a simple polygonal path γ. Bridge trees are similar to data structures used by Overmars and van Leeuwen [OL] and Guibas and Hershberger [GH]. Bridge trees exploit the observation that if paths γ and γ' are disjoint, then the hulls $h(\gamma)$ and $h(\gamma')$ have at most two common tangents [CG2]. (This is a crucial observation for our algorithm, and is the reason for the tree-untangling step of Section 3.4.) If γ can be split into γ_l and γ_r by erasing an edge, then $h(\gamma)$ is either identical to one of $h(\gamma_l)$ and $h(\gamma_r)$, or its boundary consists of a convex chain from $h(\gamma_l)$, a convex chain from $h(\gamma_r)$, and two outer common tangents joining the chains. The bridge tree for γ stores the two tangents explicitly and represents the chains by pointers to $\mathscr{B}(\gamma_l)$ and $\mathscr{B}(\gamma_r)$.

To find the bridge tree for Γ, we remove an edge from the middle of Γ to get the subpaths Γ_l and Γ_r, then build $\mathscr{B}(\Gamma)$ from the recursively constructed $\mathscr{B}(\Gamma_l)$ and $\mathscr{B}(\Gamma_r)$. An arbitrary subpath γ of Γ can be expressed as the concatenation of $O(\log n)$ of the paths produced during the construction of $\mathscr{B}(\Gamma)$. We use the bridge trees for these paths to find $\mathscr{B}(\gamma)$.

Let us state the properties of bridge trees more formally.

(1) We can build a bridge tree $\mathscr{B}(\Gamma)$ for Γ in $O(n)$ time and space.
(2) Given $\mathscr{B}(\Gamma)$, we can extract $\mathscr{B}(\gamma)$ for any subpath γ of Γ in $O(\log^2 n)$ time.

Given a bridge tree $\mathscr{B}(\gamma)$, we can in $O(\log n)$ time

(3) find the tangents to $h(\gamma)$ that pass through a given point,
(4) find the extreme vertex of $h(\gamma)$ in a query direction,
(5) find the intersections of $h(\gamma)$ with a line, and
(6) count the number of hull edges between two vertices of $h(\gamma)$.

We can also

(7) list the k edges of $h(\gamma)$ between two hull vertices in $O(\log n + k)$ time.

Given $\mathscr{B}(\gamma)$ and $\mathscr{B}(\gamma')$ for disjoint paths γ and γ', we can

(8) find the inner common tangents of $h(\gamma)$ and $h(\gamma')$ in $O(\log n)$ time, if they exist, and
(9) find the outer common tangents of $h(\gamma)$ and $h(\gamma')$ in $O(\log^2 n)$ time, if $h(\gamma)$ and $h(\gamma')$ intersect in at most two points and we are given vertices on $h(\gamma)$ and $h(\gamma')$ that lie on $h(\gamma \cup \gamma')$.

Our representation of the hull $H = h(\gamma_1 \cup \cdots \cup \gamma_s)$ uses bridge trees as building blocks. The hull H consists of chains from the hulls of the γ_i's alternating with tangent edges that link the chains. Between the endpoints of a chain from a single hull $h(\gamma_i)$, the hull H is identical to $h(\gamma_i)$. We represent each $h(\gamma_i)$ by its bridge tree $\mathcal{B}(\gamma_i)$; we represent H as a circular list of tangent edges alternating with pointers to these bridge trees. We call this structure a *necklace*: we think of the bridge trees as beads strung on a loop of tangent edges. A necklace built from hulls of s disjoint simple paths has size $O(s)$, exclusive of the bridge trees to which it points. This follows from the fact that any two convex hulls have at most two intersections and from a simple Davenport-Schinzel argument. If we store the tangent edges in a binary search tree augmented with hull edge counts, the resulting hybrid structure supports operations (3)-(9) listed above.

We use $\mathcal{B}(\Gamma)$ to produce $\mathcal{B}(\gamma_1), \ldots, \mathcal{B}(\gamma_s)$ in $O(s \log^2 n)$ time. In the course of calculating $h(\gamma_1 \cup \cdots \cup \gamma_s)$, we need to use operation (9) to find the tangents of certain pairs of hulls. The following lemma shows how to find the special vertices that operation (9) requires.

Lemma 4.1. *Let γ and γ' be disjoint simple polygonal paths whose convex hulls are represented by $\mathcal{B}(\gamma)$ and $\mathcal{B}(\gamma')$. If we are given a ray r that originates at a vertex of γ and does not intersect γ', we can determine whether γ lies inside the hull of γ' in $O(\log n)$ time. If γ is not contained in $h(\gamma')$, we can in $O(\log n)$ time find a vertex of γ that lies on $h(\gamma \cup \gamma')$.*

Proof. If γ lies inside $h(\gamma')$, it must lie in some bay bounded by γ' and an edge e of $h(\gamma')$. In this case r must intersect e. If γ is not fully contained in $h(\gamma')$, r may or may not intersect the hull of γ'. If r intersects $h(\gamma')$, let e be the hull edge it cuts. An edge of γ must cross e to get out of $h(\gamma')$, and hence some vertex of γ lies on the opposite side of e from γ'. Thus γ is contained in $h(\gamma')$ if and only if r intersects an edge e of $h(\gamma')$ and $h(\gamma)$ lies on the same side of e as γ'. Since bridge trees allow each of the operations necessary to test this condition to be performed in $O(\log n)$ time, the first claim follows.

If γ is not fully contained in $h(\gamma')$, we want to find a vertex of γ on $h(\gamma \cup \gamma')$. If r intersects $h(\gamma')$, let e be the edge it cuts; if r does not intersect $h(\gamma')$, define e to be a tangent edge from the origin of r to $h(\gamma')$. In both cases, the vertex of $h(\gamma)$ where the tangent line is parallel to e and where γ and γ' lie on the same side of the tangent is on $h(\gamma \cup \gamma')$. Using operations (3)-(5), we can find this vertex in $O(\log n)$ time. □

This lemma lets us find the common tangents (if any) of $h(\gamma_i)$ and $h(\gamma_j)$ for any i and j. The edges of Γ cut by our dual query line l provide the rays that the lemma requires: At least one endpoint of γ_i belongs to an edge of Γ cut by l. The edge can be extended to infinity on the other side of l without hitting γ_j, and hence can be used in Lemma 4.1. Using operation (9), we can find the desired common tangents, if they exist. We use this fact in the proof of the main result of this section:

Theorem 4.2. *Given an untangled spanning tree T of the n points in P, we can preprocess it in linear time to answer halfplane convex-hull queries. Given a query halfplane l^*, we can compute a necklace representation of $h(P \cap l^*)$ in $O(\hat{s} \log^3 n)$ time, where $\hat{s}$ is the stabbing number of l with respect to T.*

Proof. The preprocessing phase computes a path Γ from the given spanning tree, then builds the bridge tree $\mathcal{B}(\Gamma)$. At query time we find the subpaths $\gamma_1, \ldots, \gamma_s$ in $l^* \cap \Gamma$ [CG2], then build $\mathcal{B}(\gamma_1), \ldots, \beta(\gamma_s)$ in $O(\hat{s} \log^2 n)$ time. (Note that $\hat{s}$ is between $2s-2$ and $2s$.) We combine $h(\gamma_1), \ldots, h(\gamma_s)$ into a single hull using a divide-and-conquer approach.

We find the necklace for $h(\gamma_1 \cup \cdots \cup \gamma_s)$ by first partitioning the set $\{\gamma_1, \ldots, \gamma_s\}$ into two subsets of equal or nearly equal size, next finding the necklace for each subset separately, and finally merging the two necklaces into a single necklace. We must describe the merging step.

We split each necklace at its leftmost and rightmost vertices, then merge upper and lower necklaces separately. We merge two upper necklaces using a left-to-right sweep. For each upper necklace, the endpoints of the tangent edges specify a sequence of x-coordinates. We merge the sequences for the two necklaces to get a single sequence with k elements $x_1, x_2, \ldots, x_k$, for some $k = O(s)$. See Fig. 6. In any interval (x_i, x_{i+1}), each necklace has just one element, either a piece of a subpath hull or a tangent edge. We scan through these intervals from left to right. In each interval we find the joint upper hull of the two elements in $O(\log^2 n)$ time using Lemma 4.1 and operations (3), (5), and (9). We find the complete upper hull of the original two necklaces using a Graham scan [PS] through the intervals, at each step finding the common tangent from the hull in the current interval to the convex hull to its left. Operation (9) is applied $O(k)$ times during the scan, and thus the scan takes $O(k \log^2 n)$ time. The resulting hull is a list of subpaths and common tangents between them, but the hull may be split unnecessarily at interval endpoints. In $O(k)$ time we can merge adjacent hull pieces that come from the same subpath or tangent edge. This gives us the merged hull in necklace form, as desired.

Since merging two necklaces defined by a total of k subpaths takes $O(k \log^2 n)$ time, finding the necklace of all s subpaths by divide-and-conquer takes $O(\hat{s} \log^3 n)$ time. □

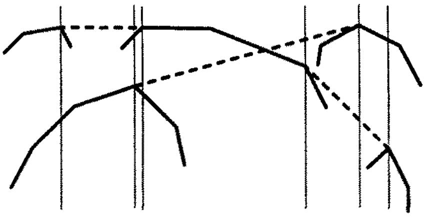

Fig. 6. Intervals arising from breaking necklaces at tangents.

Our divide-and-conquer algorithm to find the convex hull may seem like overkill: one might expect to be able to find the hull by scanning through $\gamma_1, \ldots, \gamma_s$ once, in the order given by the intersections of Γ with l. However, this simple approach is too naïve. One subpath γ_i can appear on the necklace many times. Furthermore, the order of subpath appearances along the necklace is not necessarily the same as the order given by the intersections of T with the halfplane boundary. However, it is possible that some method could use the intersection order to speed up the construction.

5. Implicit Representation of Line Arrangements

In this section we summarize how the techniques of the preceding sections can be used in combination to represent an arrangement of lines implicitly.

The first step is to dualize the set of n lines L to get a set of points P. A query point p then maps to a line l, and the face of $\mathcal{A}(L)$ that contains p maps to an hourglass shape determined by the inner common tangents of $h(P \cap l^+)$ and $h(P \cap l^-)$, as shown in Fig. 1.

We preprocess P for finding an implicit representation of $h(P \cap l^*)$ for any halfplane $l^* \in \{l^+, l^-\}$. We use Theorem 3.5 to build a collection of $O(\log n)$ spanning trees of P with the property that, with high probability, any line l has stabbing number $\Lambda(\sqrt{n})$ with respect to at least one of them. (The actual stabbing number is $O(\sqrt{n} \log^2 n)$.) The construction uses random sampling in two places. One sample, taken from the edges of a spanning tree, is easy to pick, but the second, taken from an implicitly defined "bad" set of $O(n^2)$ lines, requires $\Lambda(n^{3/2})$ time to compute using Theorem 3.8. (Notice, by the way, that this is the only step of the algorithm that takes more than $\Lambda(n)$ time.) The spanning trees may have self-intersections; we remove these using the algorithm of Section 3.4. Finally, we build a bridge tree for the Eulerian path defined by each spanning tree, as specified in Theorem 4.2.

Given a query point p, we dualize it to get query line l, then compute necklaces for $h(P \cap l^+)$ and $h(P \cap l^-)$ based on the path Γ that intersects l the fewest times. The necklaces let us find the inner common tangents of the two hulls in $O(\log n)$ time. The endpoints of the tangents delimit a convex chain of each necklace on the side closer to l. These chains dualize to the upper and lower hulls of the face containing p. We can use the necklace search structures to calculate properties of the face, simply interpreting the dual edges and vertices as primal vertices and edges. Thus we have established the following theorem:

Theorem 5.1. *Given a set L of n lines, we can preprocess it in $\Lambda(n^{3/2})$ randomized time and $\Lambda(n)$ space so that, with high probability, for any query point p we can in $\Lambda(\sqrt{n})$ time produce an implicit representation of the face of $\mathcal{A}(L)$ that contains p. From the implicit representation we can* (1) *find the intersections of a line with the face,* (2) *find the extreme vertex of the face in any direction,* (3) *count the edges of the face, or* (4) *report the edges of the face. All the operations except* (4) *take*

$O(\log n)$ time; reporting the edges of the face takes $O(\log n + K)$ time if there are K edges.

The theorem is stated using Λ-notation; the actual bounds are $O(n^{3/2} \log^2 n)$ preprocessing time, $O(n \log n)$ space, and $O(\sqrt{n} \log^5 n)$ query time.

Pankaj Agarwal (personal communication) has observed that we can use a weaker version of our bridge-tree data structure to answer ray-shooting queries in $O(\sqrt{n} \log^4 n)$ query time, improving a special case of operation (1). A ray-shooting query specifies a point p and a direction d and asks for the first intersection of the specified ray with the arrangement of lines. To answer a query, we map it to the dual plane. In the dual, shooting a ray in direction d from point p maps to rotating a line around the dual of the ray, starting from the dual of p and stopping when the rotating line hits a dual point. Let l be the dual of point p. As described above, we find the spanning path Γ that intersects l the fewest times. The line l cuts Γ into $s = O(\sqrt{n} \log^2 n)$ fragments, each of which can be decomposed into $O(\log n)$ canonical subpaths. We do a query on each canonical subpath separately; this can be done in $O(\log n)$ time (computing a tangent to a convex set from a point). In the primal, the answer to the original shooting query is the nearest of the answers to the "subqueries." This method avoids merging bridge trees into a necklace, and so saves a logarithmic factor. The query time is $O(\log n)$ per subpath, for a total of $O(s \log^2 n) = O(\sqrt{n} \log^4 n)$.

Note that ray-shooting lets us determine whether two query points p and q lie in the same face. The line determined by p and q has the same intersections with the face containing p and the face containing q if and only if p and q lie in the same face.

6. A Tradeoff Between Storage and Query Time

In this section we present a technique that allows us to improve the query time for reporting a face in an arrangement of lines at the expense of more storage. The technique uses a random sampling step that is akin to the one used in [CEG*] for obtaining a bound on the combinatorial complexity of several cells in a line arrangement. The tradeoff between storage and query time is controlled by a parameter r, $1 \le r \le n$. If we increase storage to $\Lambda(rn)$ and preprocessing time to $\Lambda(r^{1/2} n^{3/2})$, then we can decrease query time to $\Lambda(\sqrt{n/r})$. This tradeoff carries all the way to $r \approx n$, where it corresponds to storing essentially the full arrangement of the n lines and doing a point-location in it. As we will see in the next section, another interesting point in the tradeoff occurs when $r = n^{1/3}$, in which case the query time becomes $\Lambda(n^{1/3})$.

We should also remark here that since a random sampling step is involved, the more precise statement of the tradeoff result is that we provide a search structure that (1) always works, (2) takes space $\Lambda(rn)$, and (3) with high probability takes preprocessing time $\Lambda(r^{1/2} n^{3/2})$ and can answer any face query in time $\Lambda(\sqrt{n/r})$.

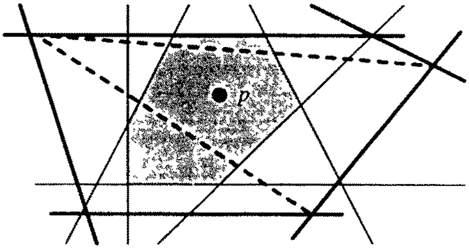

Fig. 7. The face containing p crosses a triangle.

Our technique proceeds as follows. We select at random a sample of size r among the given lines and triangulate all its faces by drawing auxiliary edges, if necessary. The ε-net theory of Haussler and Welzl [HW] (or the probabilistic lemma of Clarkson [C]) then guarantees that with high probability each of the $O(r^2)$ triangles thus created is intersected by at most $O(n \log r/r)$ of the arrangement lines. This is only slightly more than $O(n/r)$, the expected number of lines that cut through a triangle. See [CEG*] for more details on such arguments. Now the idea of the tradeoff technique is to treat all the lines cutting through a triangle as a subproblem to be solved by the methods of Section 4.

Consider a specific triangle T and the collection of all original lines cutting it (strictly speaking, cutting its interior). For a query point p, if p lies in T, then we want the subproblem for T to return to us the face containing p. However, the face containing p in the subproblem might not be identical to the face containing p in the full arrangement. There are two possible reasons for this: either the face in the full arrangement is bordered by a line on the boundary of T, or triangles bordering T also contribute sides to the face, as Fig. 7 shows.[4] These problematic faces are exactly those that touch (share an edge with) a sample line, or are crossed by an auxiliary edge of the triangulation. To deal with these faces, we just precompute and store them all in an explicit subdivision S of the plane. We then preprocess S for efficient point location by one of the standard techniques. Now, when a query point p is given, we first do a point location for p in S. If p falls into one of the convex faces of S corresponding to faces of the original arrangement of the n lines, then we report that face and we are done. Otherwise p must lie in a face of S that is fully covered by a unique triangle T of the triangulation. Furthermore, in that case the face of p in the subproblem is also the face of p in the full arrangement. So by an application of the methods of Section 4 we can complete the computation.

What is the cost of all this? The cost of choosing the random sample of the lines and triangulating the faces of their arrangement is $O(r^2)$. It is known [CGL], [EOS] that in an arrangement of n lines the combinatorial complexity of all faces touching a particular line (the *horizon* or *zone* of the line [E]) is $O(n)$. The results of [EGP*], [EGS1] imply that these faces can actually be calculated in time $\Lambda(n)$. Thus the faces touching all the sample lines can be computed in time $\Lambda(rn)$ and stored in space $O(rn)$. The faces crossed by the auxiliary edges in the triangulation can be obtained in a similar fashion, by combining the horizons corresponding to the lines supporting these edges in each of the relevant subproblem arrangements. Since there are only $O(n \log r/r)$ lines in each triangle subproblem, a total of $O(r^2)$ triangles, and at most two auxiliary edges bordering each triangle, we conclude that the horizons of all auxiliary edges can be computed in $\Lambda(rn)$ time and stored in $\Lambda(rn)$ space.[5] Finally, since the point-location structure takes linear space in the size of the underlying subdivision S, the overall space that we need is $\Lambda(rn)$ for that structure and also $\Lambda(rn)$ in total for the triangle subproblems.

---

[4] In fact the face containing p in the full arrangement can cut across many such triangles.

Theorem 6.1. *For each r, $1 \leq r \leq n$, we can build a structure for the face-reporting problem that takes $\Lambda(rn)$ space and which, with high probability, can be built in time $\Lambda(r^{1/2}n^{3/2})$ and allows us to answer any face query in time $\Lambda(\sqrt{n/r}) + O(K)$, where K is the size of the reported face.*

7. Implicit Faces in Arrangements of Segments

In this section we extend our techniques to collections of line segments, as opposed to infinite lines. In this situation many new difficulties arise: the faces we may wish to report need not be convex, or even simply connected—see Fig. 8. Our previous implicit techniques cannot be extended easily, as we have made very heavy use of convexity. Instead, our approach will use the method of partitioning into "triangular" subproblems, as developed in the previous section. The reason partitioning helps is that in a "small" triangle T we can hope that most segments intersecting T actually go all the way through T. Since we are interested only in faces clipped to within T, when we solve the subproblem associated with T we can treat these long segments as infinite lines. A similar

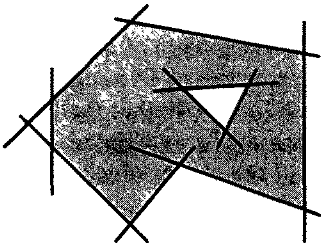

Fig. 8. A face in an arrangement of segments that is neither convex nor simply connected.

---

[5] In this description we have skipped over a number of straightforward details that are left to the reader.

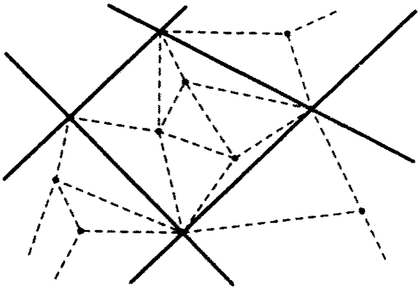

Fig. 9. We triangulate the arrangement of sampled lines $\mathscr{A}(L')$ together with the sampled points G'.

idea has been used in [GOS] to solve the problem of counting or reporting line-segment intersections.

Let L denote the set of lines supporting our n given segments, and G denote the set of endpoints of these segments. As in the previous section, we now use a random-sampling technique to partition our problem into subproblems. Each subproblem is associated with a triangle T and consists of all the original segments cutting through T. Let r be an integer parameter to be fixed later. We choose a random sample L' of r of the lines in L and, independently, a random sample G' of r^2 of the points in G. We now triangulate the arrangement of the lines in L' with the additional points of G' thrown in—see Fig. 9. By using $O(r^2)$ triangles we can obtain a triangulation such that no triangle contains one of the points in G' in its interior, or has its interior intersected by one of the lines in L'. Now the ε-net theory [HW], [C] allows us to conclude that, with high probability, no triangle is intersected by more than $O(n \log r/r)$ of the lines in L, *and* no triangle contains more than $O(n \log r/r^2)$ of the points in G. Notice that in this step the points in G are partitioned among the triangles, while each line in L may be distributed to many triangles. Thus, on the average, this partition step will decrease the number of points in a triangle more than it will decrease the number of lines.

We now construct and store all faces of the original arrangement of n segments that are bordered or crossed by one of the r sample lines or by one of the auxiliary $O(r^2)$ triangulation edges. These faces belong either to the r *horizons* or *zones* [E] of the sample lines in our arrangement of n segments, or to the $O(r^2)$ horizons of the auxiliary edges in arrangements of $O(n \log r/r)$ segments each. The results of [EGP*], [EGS1] can be used to prove that the total complexity of these faces is only $\Lambda(rn)$, in that they can all be calculated in $\Lambda(rn)$ time. All these faces together define a subdivision of the plane, which we preprocess for efficient point location. The overall time and storage cost of this step is $\Lambda(rn)$.

Our query algorithm then begins by doing a point location in the above structure. If the query point p happens to lie in one of the explicitly stored faces, then we can complete processing the query without having to use any additional structures. Otherwise p falls into a region that is fully covered by a unique triangle T of the triangulation and has the property that the face containing p in the subproblem corresponding to T is exactly the face of p in the full arrangement.

How do we solve these triangle subproblems? Let us concentrate on a particular triangle T. The set of lines L_T consists of the lines supporting the segments cutting T, and the set of points G_T consists of the endpoints of these segments that actually lie in T. It is useful to categorize the segments crossing T into two groups: the "long" segments that cut all the way through T, and the "short" segments that have an endpoint in T. Notice that there can be no more short segments in T than the number of points in G_T. If we choose $r \approx n^{1/3}$, then the ε-net bounds imply that that the number of lines in L_T (and therefore the number of long segments) is at most $O(n^{2/3} \log n)$, while the number of points in G_T (and therefore the number of short segments) is at most $O(n^{1/3} \log n)$. The fact that the bound on the number of short segments is only slightly greater than the square root of the bound on the number of long segments is favorable for us, as we will now see. This consideration determined the choice of r made above.

For the triangle T we store two separate structures: the full arrangement of the short segments of T, and the data structure of Theorem 5.1 for the duals of the infinite lines supporting the long segments of T. Because of our choice of r, the total size of these two structures is only $\Lambda(n^{2/3})$. Summed over all $O(r^2)$ triangles, the total is $\Lambda(n^{4/3})$. In combination with the point-location structure described above, the overall storage that we use is $\Lambda(n^{4/3})$; the preprocessing cost is, with high probability, $\Lambda(n^{5/3})$.

To answer a query in T, we separately solve it in the (explicit) arrangement of the short segments in T, and in the (implicit) arrangement of the long segments in T; then we combine the results. It is easy to see how this works for the types of queries we have been concerned with, except for face reporting, where we encounter a technical difficulty: the two subfaces obtained in the two separate arrangements can have complexities much greater than that of the desired face, which is a connected component of their intersection.

There is, however, a lot of structure in the situation that we can exploit. Recall that what we desire is to answer the query at a cost which is $\Lambda(n^{1/3}) + O(K)$, where K is the output size. By the results in [EGP*], the complexity of the "short" face is only $\Lambda(n^{1/3})$, so we can afford to look at all of it. Notice also that the implicitly represented "long" face is convex.

To deal with the above difficulty, we can start out by triangulating the short face. Then we intersect separately each of these triangles with the long face. A triangle can be intersected efficiently with the long face: we just intersect the long face with each of the lines defining the triangle and collect the appropriate pieces. By the results of Section 4, we can construct an implicit representation of the portion of the long face inside each triangle in $\Lambda(1)$ time per triangle. We can now put all the separate intersections together into the true intersection of the long and short faces. Notice that the additional breakage introduced along the long face by the edges used for triangulating the short face is only $\Lambda(n^{1/3})$ (two breakpoints per diagonal), so it can be paid out of the allowable overhead. This computation might actually produce several intersection faces, not just the one containing the query point p. The others, however, can be easily disposed of if we keep around the dual graph of the triangulation of the short face. We can start at the node v of this graph corresponding to the query point p and in a depth-first traversal from v collect all the pieces corresponding to the final face. Once we have collected these pieces, we expand their implicit representations to get an explicit description of the desired component of the intersection of the long and short faces. The expansion takes $\Lambda(n^{1/3})+O(K)$ time, where K is the size of the final face. This completes our argument.

Theorem 7.1. *Given n segments in the plane, then in* $\Lambda(n^{4/3})$ *space we can construct a data structure such that, with high probability, the preprocessing cost is* $\Lambda(n^{5/3})$, *and, for every query point p, the face of the arrangement of these segments containing p can be reported in time* $\Lambda(n^{1/3})+O(K)$, *where K is the size of the desired face.*

8. Reporting Many Faces

From the results of [CEG*] it is known that the complexity of m distinct faces in an arrangement of n lines in the plane is $O(m^{2/3}n^{2/3}+n)$. The method used in that paper is based on a random-sampling step like the one presented in the previous section. If we think of the m desired faces as being specified by m given points, then the random-sampling step selects r lines and r^2 points and triangulates the arrangement of these lines and points. Again by the ε-net property, each triangle subproblem has no more than $O(n \log r/r)$ lines and no more than $O(m \log r/r^2)$ points. The proper value of r to use here is $r=m^{2/3}/n^{1/3}$ because that makes the bound on the number of points per triangle $\Lambda(n^{2/3}/m^{1/3})$, which, just as before, is roughly the square root of the bound on the number of lines per triangle, $\Lambda(n^{4/3}/m^{2/3})$.

Suppose now that we use our face-reporting technique individually for each point in each triangle subproblem. However, since in each face we already know the queries we will make, we can use the technique of Theorem 3.4 to build a *single tree* that is good for the prespecified queries on the average. Thus we can avoid the extra cost associated with Theorem 3.8; with high probability, it takes only $\Lambda(n^{4/3}/m^{2/3})$ time to construct a structure that allows face reporting with $\Lambda(n^{2/3}/m^{1/3})$ overhead on the average for each face query. The overall cost per subproblem is $\Lambda(n^{4/3}/m^{2/3})+O(K)$, where K is the total size of the faces to be reported. If we sum this over all triangles and in addition take into consideration the cost of the point-location structure implied by the partitioning step, we obtain the following result on the cost of reporting m faces in an arrangement of n lines.

Theorem 8.1. *Given m points lying in distinct faces of an arrangement of n lines, then, with high probability, in time* $\Lambda(m^{2/3}n^{2/3}+n)$ *and space* $O(m^{2/3}n^{2/3}+n)$, *we can report the faces containing these points.*

This result matches the worst-case output complexity [CEG*] to within a polylogarithmic factor.

References

[AS] B. Aronov and M. Sharir. Triangles in space, or building and analyzing castles in the air. In *Proceedings of the 4th ACM Symposium on Computational Geometry*, pages 381–391, ACM, June 1988.

[BO] J. L. Bentley and T. A. Ottman. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, 28(9):643–647, 1979.

[B] K. Q. Brown. Geometric Transforms for Fast Geometric Algorithms. Ph.D. thesis, Carnegie-Mellon University, 1980.

[CE] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 590–600, IEEE, 1988.

[CG1] B. Chazelle and L. Guibas. Visibility and intersection problems in plane geometry. In *Proceedings of the ACM Symposium on Computational Geometry*, pages 135–146, ACM, 1985. Submitted to *Discrete and Computational Geometry*.

[CG2] B. Chazelle and L. J. Guibas. Fractional cascading: II. Applications. *Algorithmica*, 1:163–191, 1986.

[CGL] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[CW] B. Chazelle and E. Welzl. Range searching and VC-dimension: a characterization of efficiency. 1988. Manuscript.

[C] K. Clarkson. New applications of random sampling in computational geometry. *Discrete and Computational Geometry*, 2:195–222, 1987.

[CEG*] K. Clarkson, H. Edelsbrunner, L. Guibas, M. Sharir, and E. Welzl. Combinatorial complexity bounds for arrangements of curves and surfaces. In *Proceedings of the 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 568–579, IEEE, 1988.

[E] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*. EATCS Monographs on Theoretical Computer Science, Volume 10, Springer-Verlag, Berlin, 1987.

[EG] H. Edelsbrunner and L. J. Guibas. Topologically sweeping an arrangement. In *Proceedings of the 18th ACM Symposium on Theory of Computing*, pages 389–403, ACM, May 1986.

[EGP*] H. Edelsbrunner, L. Guibas, J. Pach, R. Pollack, R. Seidel, and M. Sharir. Arrangements of curves in the plane: Topology, combinatorics, and algorithms. In *Proceedings of the 15th ICALP*, 1988.

[EGS1] H. Edelsbrunner, L. J. Guibas, and M. Sharir. The complexity of many faces in arrangements of lines and of segments. In *Proceedings of the 4th ACM Symposium on Computational Geometry*, pages 44–55, ACM, June 1988.

[EGS2] H. Edelsbrunner, L. Guibas, and J. Stolfi. Optimal point location in a monotone subdivision. *SIAM Journal on Computing*, 15:317–340, 1986.

[EOS] H. Edelsbrunner, J. O'Rourke, and R. Seidel. Constructing arrangements of lines and hyperplanes with applications. *SIAM Journal on Computing*, 15:341–363, 1986.

[EW] H. Edelsbrunner and E. Welzl. Halfplanar range search in linear space and $O(n^{0.695})$ query time. *Information Processing Letters*, 23:289–293, 1986.

[GH] L. Guibas and J. Hershberger. Optimal shortest path queries in a simple polygon. In *Proceedings of the 3rd ACM Symposium on Computational Geometry*, pages 50–63, ACM, June 1987.

[GHS] L. Guibas, J. Hershberger, and J. Snoeyink. Bridge trees: a data structure for convex hulls. 1989. In preparation.

[GOS] L. Guibas, M. Overmars, and M. Sharir. Intersecting line segments, ray shooting, and other applications of geometric partitioning techniques. In *Proceedings of the First Scandinavian Workshop on Algorithm Theory*, pages 64–73. Lecture Notes in Computer Science, Volume 318, Springer-Verlag, Berlin, 1988.

[HW] D. Haussler and E. Welzl. Epsilon-nets and simplex range queries. *Discrete and Computational Geometry*, 2:127–151, 1987.

[Ki] D. Kirkpatrick. Optimal search in planar subdivisions. *SIAM Journal on Computing*, 12:28-35, 1983.

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

[M] J. Matoušek. Spanning trees with low crossing number. 1988. Manuscript.

[OL] M. Overmars and H. van Leeuwen. Maintenance of configurations in the plane. *Journal of Computer and System Sciences*, 23:166-204, 1981.

[PS] F. P. Preparata and M. I. Shamos. *Computational Geometry*. Springer-Verlag, New York, 1985.

[ST] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29:669-679, 1986.

[SH] M. I. Shamos and D. Hoey. Geometric intersection problems. In *Proceedings of the 17th IEEE Symposium on Foundations of Computer Science*, pages 208-215, IEEE, 1976.

[We] E. Welzl. Partition trees for triangle counting and other range searching problems. In *Proceedings of the 4th ACM Symposium on Computational Geometry*, pages 23-33, ACM, June 1988.

[Wi] D. E. Willard. Polygon retrieval. *SIAM Journal on Computing*, 11:149-165, 1982.

Received July 22, 1988, and in revised form March 27, 1989.