Lecture V
PLANE GRAPHS AND POINT LOCATION

Euler's formula relating number of vertices, edges and faces of a plana embedding of graph is a fundamental relation in computational geometry. It leads to all the favorable computational properties of planar graphs, as compared to general graphs. We introduce the notion of skeletons and their computation. In general, we need the language of cell complexes as in topology where skeletons might be called 1-complexes.

The problem of point location, Kirkpatrick's elegant solution, and the alternative of Seidel is treated.

§1. Plane Graphs and Skeletons

Let G be a undirected graph $G = (V, E)$. Let the map μ assign to each $v \in V$ a distinct point $\mu(v)$ in the plane. The points are distinct in the sense that $u \neq v$ implies $\mu(u) \neq \mu(v)$. This map extends naturally to edges where $e = (u, v) \in E$ is mapped to the open line segment $\mu(e) = (\mu(u), \mu(v))$. This extended map μ is called a **linear plane embedding** of G if the following two conditions hold.

- For all edges $e \neq f$, $\mu(e) \cap \mu(f) =$,
- If v is not an endpoint of e, then v does not lie in the closure of $\mu(e)$.

Not all graphs admit such a embedding; those that do are called **planar graphs**. We will simply say "embedding" instead of "linear plane embedding". By a **plane graph** we mean[1] a planar graph G together with some linear plane embedding μ.

Let S be a set of open line segments (called "edges") and points ("vertices"). We call S a **skeleton** if it satisfies the following properties:

- For all edges $e, f \in S$, $e \neq f$ implies $e \cap f =$.
- Let $e \in S$ be an edge. If a vertex $v \in S$ lies in the closure of e then v is an endpoint of e. Conversely, if v is an endpoint of e then $v \in S$.

Clearly, if μ is an embedding of G, then the set

$$\{\mu(v) : v \in V\} \cup \{\mu(e) : e \in E\}$$

is a skeleton. Conversely, every skeleton is the embedding of a graph which we denote by $G(S) = (V(S), E(S))$. By abuse of language, we sometimes call S an embedding of $G(S)$.

Given any graph G, we have the usual numerical quantities $|V|, |E|$ and the number of connected components $\beta(G)$. We will call the quantities $|V|$, $|E|$ and the number of connected components of G the **embedding invariants** of G, and denote them by

$$\nu_0(G), \nu_1(G), \beta(G),$$

respectively. If S is a skeleton, we will write $\nu_0(S)$ for $\nu_0(G(S))$, etc. But we can define another quantity for S, *the number of regions* induced by S. More precisely, consider the set $K = \mathbb{R}^2 \setminus \cup S$. The set K partitions

[1] This technical distinction between "planar graph" and "plane graph" is possibly confusing in ordinary speech.

naturally into a finite number of maximally connected open sets, each of which is called a **region** of the embedding. Note that K has exactly one unbounded region. Let $\nu_2(S)$ denote the number of regions. It is less clear that $\nu_2(S)$ is an embedding invariant of $G(S)$. This is however true, and will be proved below. Hence we may write $\nu_2(G)$ for $\nu_2(S)$ where S is any embedding of G. Note that $\nu_2(G)$ is only defined when G is planar.

Remark: Two closed line segments are said to be **crossing** if their relative interios intersect; otherwise they are non-crossing. Note that non-crossing line segments may share endpoints. Then a skeleton S alternatively be represented by a set of closed line segments that are non-crossing.

§2. Euler's Formula for Embeddings

A skeleton S (or graph G) can be incrementally constructed via a sequence of $|S|$ (or $|V|+|E|$) operations, where each operation comes under one of the following two types:

- Add an isolated point (a vertex).
- Add an open line segment (an edge) connecting two existing isolated points (vertices).

More precisely, a **construction sequence** for S is

$$S_0 \subseteq S_1 \subseteq S_2 \subseteq \cdots \subseteq S_m \tag{1}$$

where $S_0 = \emptyset$, $S_m = S$, and for $i \geq 0$, S_{i+1} is obtained from S_i using one of our operations above. Clearly, $|S_i| = i$ for $i = 0, \ldots, m$.

Theorem 1 *The following formula of Euler holds for any skeleton S:*

$$\nu_0(S) - \nu_1(S) + \nu_2(S) = 1 + \beta(S). \tag{2}$$

Proof. We use induction on $|S|$. The formula is true when S is the empty set:

$$0 - 0 + 1 = 1 + 0.$$

For $|S| > 0$, consider any construction sequence (1) that ends in S. Let S' be the skeleton just before S in the sequence (so $S' = S_{m-1}$). By induction hypothesis, Euler's formula holds for S'. So, we need to show that adding a vertex or an edge preserves the invariant. When we add a vertex, the only quantities in the Euler's formula (2) that change are the number of vertices and the number of connected components of the graph: both increases by 1,

$$\nu_0(S) = \nu_0(S') + 1, \beta(S) = \beta(S') + 1.$$

But these changes preserve our Euler's formula. When we add an edge, there are two possibilities: (i) the two endpoints of the edge belong to disjoint components of the graph, or (ii) they belong to a common component. In case (i), the quantities that change are

$$\nu_1(S) = \nu_1(S') + 1, \beta(S) = \beta(S') - 1.$$

Thus, we gained an edge but lost one connected component. This preserves the Euler's formula (2). In case (ii), the changes are

$$\nu_1(S) = \nu_1(S') + 1, \nu_2(S) = \nu_2(S') + 1.$$

Thus, we gained an edge as well an a region (one of the original regions split into two). Again, Euler's formula continues to hold. **Q.E.D.**

The above proof apparently depends on a choice of a sequence (1), namely, on the order in which vertices and edges are introduced. But the numerical quantity $\nu_2(S)$ depends only on the final set S, and not on the sequence of operations to reach S. We now show that $\nu_2(S)$ depends only on the graph $G(S)$.

Corollary 2 *The number $\nu_2(G)$ is well-defined for a planar graph G.*

Proof. Let S be any embedding of G. We know that $\nu_0(S), \nu_1(S), \beta(S)$ are functions of G alone, independent of the choice of S. Thus Euler's formula (2) for S implies that $\nu_2(S)$ is also a function of G alone. **Q.E.D.**

Remark: We can equally develop this section by defining the "dual" of a construction sequence: We may define a **destruction sequence** for any embedding S to be a sequence $(S_0, S_1, \ldots, S_m)$ where $S_0 = S$, and each S_{i+1} is obtained from S_i by removing an arbitrary edge or by removing a vertex that is no longer incident on any edge.

§3. Consequences of Euler's Formula

Let $G = (V, E)$ be a planar graph. Our first goal is to bound the **average degree** $\delta = \delta(G)$ of vertices in a planar graph:

$$\delta = \frac{1}{\nu_0} \sum_{v \in V} d(v)$$

where $d(v)$ is the degree of v, *i.e.*, number of edges in E incident on v. Now $\sum_{v \in V} d(v)$ can be regarded as counting the number of (vertex, edge) incidences *from the viewpoint of the vertices*. We can count also these incidences from *the view point of the edges*, and this number is $2\nu_1$ since each edge is involved in two such incidences. It follows that

$$\delta = \frac{2\nu_1}{\nu_0}. \tag{3}$$

In combinatorics, whenever we can count a quantity in two independent ways, we have a non-trivial relation such as (3). Good, let us do this again. Why not count the number of (edge, region) incidences. The difference is that we now only get bounds (upper or lower) on the desired quantity. From the viewpoint of the edges, this count is exactly $2\nu_1$. Actually, we have a technicality here: sometimes, an edge may have the same region on both of its sides – do we count this as one or two incidences? If we count this as one (edge, region) incidence, then $2\nu_1$ is not exact, but an upper bound. But it turns out not to matter. From the viewpoint of the regions, we get a lower bound of $3\nu_2$ on the number of incidences. This lower bound is the first time we actually exploit the fact that our embedded edges are *linear* curves, so that each region is bounded at least three edges. Thus we have

$$2\nu_1 \geq 3\nu_2. \tag{4}$$

Thus the number of regions ν_2 is at most $2\nu_1/3$. Plugging this inequality into Euler's formula to eliminate ν_2, we obtain

$$\begin{aligned} \nu_0 - \nu_1 + \frac{2\nu_1}{3} &\geq 1 + \beta \\ \nu_0 &\geq 1 + \frac{\nu_1}{3}. \end{aligned}$$

Thus the average degree is less than 6:

$$\frac{2\nu_1}{\nu_0} < \frac{2\nu_1}{\nu_1/3} = 6.$$

But at most half of the vertices has twice the average degree or more. Thus we have shown: *In a plane graph G with n nodes, at least $\lfloor n/2 \rfloor = \lfloor \nu_0(G)/2 \rfloor$ of its vertices has degree less than* 12.

Let us summarize: In a plane graph with v vertices and e edges and r regions,

$$r \le 2e/3, \qquad e \le 3v.$$

Thus $r \le 2v$. Morever, at least $\lfloor v/2 \rfloor$ of the vertices have degree at most 11.

§4. Data Structures for Subdivisions

The information represented by an embedding (or a skeleton) needs to be rationally encoded to support algorithms for such embeddings. Several data structures have been proposed in the literature, and we now examine some of them.

The Problem with Holes. First some terminology: if f, f' are two adjacent faces, with f' contained in the closure of f, then we say that f' **bounds** f. The dimension of f' will be less than the dimension of f. The boundary of a region will in general be comprised of several **boundary components**, which are pairwise disjoint simple polygons (=vertices and edges), For bounded regions, there is a **distinguished** boundary component, namely, the the outermost boundary. The non-distinguished boundary components can be seen to bound holes. Connected regions that do not have holes are said to be **simply-connected**. Alternatively, for a non-empty skeleton, a region is simply connected iff it is bounded and has one boundary component.

One way to convert bounded regions into simply-connected regions is to introduce an edge to connect the boundary to each hold to the rest of the boundary. Such an edge (called an **isthmus**) is characterized by having the same region on both sides of the edge. For simplicity, we often assume that the bounded regions are simply-connected.

Requirements. Let S be a skeleton. What do we require from a data structures $D(S)$ for $\Pi(S)$? Basically, we expect to search from any face of $\Pi(S)$ its adjacent faces quickly.

- Given a region R in $\Pi(S)$, we expect to be able to visit each of its bounding edges and vertices in some circular order at constant cost per edge or vertex.
- Given an edge e, we expect to locate the two incident regions and two incident vertices in constant time.
- Given a vertex v, we expect to be able to visit each of the edges and regions that it bounds in some circular order, at constant cost per edge or region.

Let us give an example of an operation that does not automatically fall out of such representations: given a vertex, what is the edge (if any) that is vertically above it? This query cannot be obtained in constant time using the above information.

First Attempt: Augmented Adjacency Lists. To indicate that the correct data structure is not entirely trivial, let us explore the reasonable suggestion to represent $\Pi(S)$ by any efficient representation of the graph $G(S)$, augmented by additional information that arises from the embedding. One of the most useful representations of graphs is the **adjacency list** representation: we have a vertex list and an edge list; for each edge in the edge list, we store its pair of endpoints, and for each vertex v in the vertex list, we store an **adjacency list** $A(v)$ comprising the edges that are bounded by v. We now propose to represent subdivisions by augmenting the adjacency list representation. The following additional information may be provided.

- We need to store a list of regions. With region, we store a list of bounding edges, one edge from each boundary component.
- With each vertex in the vertex list, we store its coordinates, (x, y),
- With each edge e in the edge list, we store the two (not necessarily distinct) regions that it bounds,
- The adjacency list $A(v)$ must now be kept in some cylic order that reflects the cyclic ordering of the edges around v.

Unfortunately, this data structure does not meet one of our above requirements. How do we traverse around the boundary of a region efficiently? The adjacency does encode information to support this traversal, but not in a form that is easily accessible. In particular, we cannot get from one bounding edge e to the "next edge" e' in constant time. It takes time proportional to the degree of the vertex that is incident to both e and e'. Nevertheless, this simple "augmented adjacency list" structure may be useful if going around the boundary of a region is not important.

An alternative solution would be to further augment our augmented adjacency list structure: store with each face, a list of all the edges bounding the face. But this is duplicating information, which is sometimes undesirable.

The DCEL Structure. There is no simple way to modify the augmented adjacency list representation above. Instead we must take it apart: the problem lies in the the monolithic "adjacency lists" (inherited from graphs). We will take the information in these lists and distribute this information among the edges. Each edge is now given an arbitrary direction. This amounts to specifying one incident vertex of e as the **start** and the other as the **stop** vertex. Relative to this direction, we have two faces called the **left** and **right faces** of e. We define the **left** and **right successors** of of e to be the two edges that are incident on the $e.stop$ and which bound the left and right (respectively) faces of e. Similarly, we can define the **left and right predecessors** of e; these are edges incident on $e.start$ and bounding the left and right (respectively) faces of e. See figure 1(a).

.

Half-Edge Data Structure. The DCEL Structure has one unsatisfactory feature, its arbitrary assignment of direction to edges. We can remove this arbitrariness by splitting each edge into two **half-edges**. The result data structure is the so-called half-edge data structure. So the two half-edges are really the same edge with opposite orientations. Such a pair is called a **twin**. For each half-edge h, let $h.twin$ be its twin. Thus, $h.twin.twin = h$. The end points of h are $h.start$ and $h.stop$ and we consider h to be directed from $h.start$ to $h.stop$. Thus $h.twin.start = h.stop$ and $h.twin.stop = h.start$. We view each half-edge as bounding a single face, denoted $h.face$. This is the face that lies on the right side of the directed half-edge. We also have $h.succ, h.pred$ refering to the half-edges that bound $h.face$ and incident on $h.stop$ and $h.start$ (respectively).

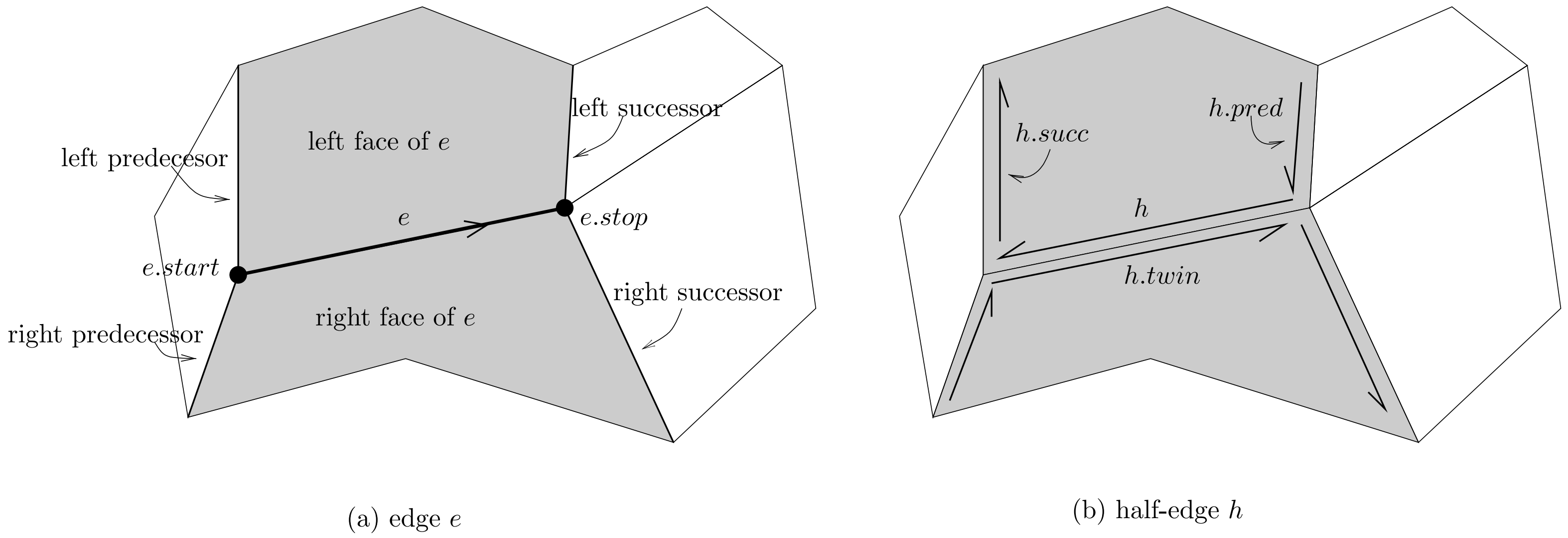


Figure 1: Successor and predecessor edges

This data structure will be our default data structure for plane subdivisions (and any general surface mesh). For this reason, let us be explicit about our conventions for this data structure.

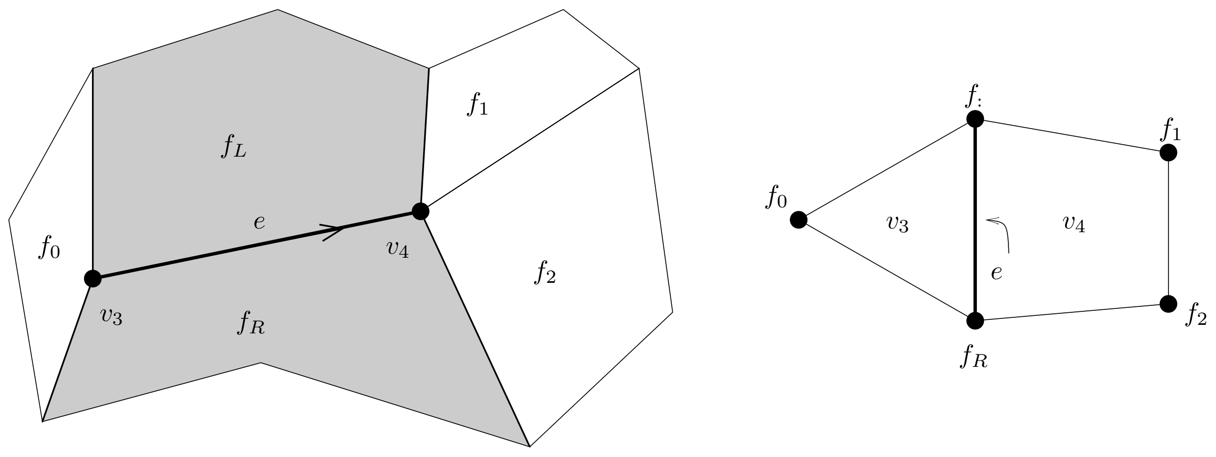
- We store three lists: vertices, half-edges and regions. Each region is assumed to he simply-connected (so has only one boundary component). We allow edges which are isthmus.
- Each half-edge h stores pointers to three half-edges: its successor $h.succ$, predecessor $h.pred$, its twin $h.twin$.
- The half-edge also points to two vertices: $h.tail$ and $h.head$. The half-edge is directed from the tail vertex to the head vertex.
- It has a pointer to the sole region that it bounds, $h.region$. This is the region which lies to its left. Following the successor pointers of a half-edge will traver the boundary of $h.region$ in a counter-clockwise manner.
- Each vertex stores its geometric coordinates (x, y), and stores a pointer to one incident half-edge.
- Each region points to one bounding half-edge.

In applications, parts of this data structure may be omitted. For instance, if it is sufficient to traverse the boundary of a face in only one direction, then we can drop the predecessor links for each half-edge. Often, we may omit the list of faces and associated pointers.

What is the complexity of this representation? When it is $O(\nu_0(S) + \nu_1(S) + \nu_2(S)) = O(n)$ where $n = \nu_0(S)$ is usually taken to be the size of this subdivision.

Variations. Some variants are called "winged edges data structures", where the "wing" terminology is suggested by the directions drawn on edges. Sometimes, redundant information may be provided to provide constant speedup. In some applications, some of the links can be omitted. As noted above, we may often drop all information related to the regions. If this is done, we may achieve significant improvement in performance (constant factor, of course). Another remark is that our skeletons allow "dangling edges" or handles which do not appear essential. But one reason to allow them is that some incremental algorithms, our data structre may pass through intermediate stages with such handles which will eventually be removed.

Quad-Edges Data Structure and Duality. The quad-edge data structure was introduced by Guibas and Stolfi. Let us now restrict attention to regions that are simply connected. Then there is a dual graph $D(G)$ in which the regions of G are vertices of $D(G)$, and the edges of $D(G)$ are still the edges of $D(G)$. In general, $D(G)$ may no longer be a simple graph – it may have multiple (or parallel) edges. See figure 2.



(a) primal (b) dual

Figure 2: Dual Graphs.

The quad-edge data structure for a skeleton S has the elegant symmetry between vertices and regions, thus giving no preferences to the graph $G(S)$ or its dual $D(G(S))$.

Remarks. For an in-depth discussion on designing a data structure for surfaces in a general geometric library, we refer to Kettner [1].

§5. Triangulation of a Subdivision

A connected subset $X \subseteq \mathbb{R}^2$ is y-monotone if every horizontal line $H(t)$ intersects X in a connected set $H(t) \cap X$. Note that if $H(t) \cap X$ is empty, it is considered a connected set. Let Π be a subdivision of the plane. We say Π is **y-monotone** if every bounded face of Π is y-monotone. We say Π is **triangulated** if bounded region is triangulated. Note that

§6. Point Location and Kirkpatrick's Structure

Let S be a skeleton. It induces a partition $\Pi(S)$ of $\mathbb{R}^2$ into disjoint sets which comprise the vertices and edges in S, as well as the regions as defined before. Thus $S \subseteq \Pi$ and the set of regions is $\Pi \setminus S$. Call $\Pi = \Pi(S)$ the called the **planar subdivision** induced by S. Each $f \in \Pi$ is called a face of the subdivision. Thus the faces are vertices (or 0-faces), edges (or 1-faces) or regions (or 2-faces).

Now assume that the underlying graph $G(S)$ is connected, *i.e.*, $\beta(G(S)) = 1$). The **planar point location problem** for S asks us to construct a data structure $D(S)$ such that for any point $q \in \mathbb{R}^2$, we can use $D(S)$ to efficiently determine the face $f(q)$ of the subdivision $\Pi(S)$ that contains q. Here q is called the **query point**.

We now describe a beautiful datastructure $D(S)$ of Kirkpatrick to solve the planar point location problem. We assume that the subdivision $\Pi(S)$ is **triangulated** (that is, each region, except for the infinite region, is a triangle). A subset $U \subseteq V(S)$ is called an **independent set** if no two vertices in U are connected by an edge. Kirkpatrick defines a sequence (hierarchy)

$$S_0, S_1, \ldots, S_h \tag{5}$$

of embeddings where S_0 is the original S, and S_{i+1} is obtained from S_i be removing an independent subset $U_i \subseteq V_i(S)$. Thus

$$V(S_{i+1}) = V(S_i) \setminus U_i.$$

We now describe $E(S_{i+1})$. This is obtained from $E(S_i)$ be removing any edge that is incident on a removed vertex of U_i, and by re-introducing edges to re-triangulate the regions that are no longer triangulated.

It is instructive to understand this re-triangulation process. Say u is a removed vertex, and as a result, we have to remove k edges that are incident on u. This create a "star-shaped region" that is centered at u with k bounding segments. To re-triangulate this region, we only need to $k-3$ new edges. Note that because we assume U is an independent set, the star-shaped region for the points in U are pairwise disjoint. Hence the re-triangulation for each u can proceed independently.

This completes the description of S_{i+1}. Intuitively, S_{i+1} is a "simplified version" of S_i. We step this simplification process when $|S_i|$ is less than some constant. We may represent each $\Pi(S_i)$ using some standard topological representation of subdivisions (e.g., half-edge data structure).

There is another important set of links in our hierarchical data structure: each face $f \in \Pi(S_{i+1})$ points to its "cause" in $\Pi(S_i)$: if f occurs as $f' \in \Pi(S_i)$, then f' is the "direct cause" of f. If not, f points to the vertex u in $\Pi(S_i)$ whose removal led to the creation of f (in this case, f is an edge or a region). Here, u the "indirect cause" of f. We have now completely describe $D(S)$, except for one addition detail – how is the independent sets U_i specified?

Let us see how we can use $D(S)$ for point location: given a query point q, we locate the face f of $\Pi(S_h)$ that contains q. In general, when we have found the face f_{i+1} in $\Pi(S_{i+1})$ that contains q, we can find the face the face f_i in $\Pi(S_i)$ that contains q as follows: follow "cause link" of f_{i+1} to some f' in $\Pi(S_i)$. If f' is a direct cause then f_i is simply f'. Otherwise, we need to search the edges and regions in $\Pi(S_i)$ that are incident on f' to find f_i. How much time does this search take? *This depends on the degree of f'.* We would like this degree to be bounded.

Complexity of Kirkpatrick's Solution. This brings us to the final detail: Kirkpatrick shows that we can choose U_i so that

(a) Each vertex in U_i has degree at most 11, and

(b) $|U_i|$ is at least $|V(S_i)|/24$.

Can we find U_i with properties (a) and (b)? This is actually a simple consequence of our bound that about half of the vertices has degree at most 11. We just pick vertices of degree at most 11 in any order, making sure that no two picked vertices are adjacent. So, each picked vertex cause us to eliminate at most 12 vertices from further consideration. Thus, we can continue this picking for at least $n/24$ times since at least $n/2$ of the nodes must be eliminated before this process halts.

Let us deduce the computational significance of (a) and (b). From (b), it follows that the hierarchy (5) is $O(\log |V(S)|) = O(\lg n)$. In fact, it is at most

$$h \le \log_{12/11} |V(S)|. \tag{6}$$

From the bound $\nu_1(S) < 3\nu_0(S)$ in the previous section, we are justified to define the **size** of S to be $n = \nu_0(S)$. From (a) we see that we can do point location for q in S_i in constant time, given that we have the answer to the query in S_{i+1}. Combined with (6), we conclude that Kirkpatrick's structure can answer queries in $O(\log n)$ time.

Discussion. Unfortunately, this beautiful data structure does not appear to be useful in practice. There are two reasons. One is that the hidden constant in the $\log n$ time performance seems to be too large. Let the time be $C \lg n$. Note that C depends linearly $1/\lg(24/23) + 11$. We will next examine more practical alternatives. The second is that the approach requires the subdivision be a triangulation. We will show how to get around this problem.

Kirkpatrick's result motivates the question: what other numbers can be used in place of the constants $(d, f) = (11, 24)$ in the above proof? It turns out that we can use $(d, f) = (9, 35/2)$.

§7. REMARKS

Dynamic Maps. See Teillaud [2].

Consistency Problem. Given a half-edge data structure, how can we verify that it is consistent? In practice, this is an important issue, since algorithms often construct erroneous structures because of numerical roundoff errors.

Exercises

Exercise 7.1: Suppose $G(S)$ is a connected graph and we want a construction sequence $(S_0, \ldots, S_m)$ for S with the property that each $G(S_i)$ is connected. In this case, we replace the operation of adding an isolated vertex by an operation that simultaneously adds an isolated vertex *and* an edge that connects it to the rest of the graph. Write a constructive geometry package supporting this set of operations. ♢

Exercise 7.2: Give a detail accounting of the space needs for our half-edge data structure. Compare this to the quad-edge and the DCEL data structures. ♢

End Exercises

References

[1] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry: Theory and Applications*, 13:65–90, 1999.

[2] M. Teillaud. Union and split operations on dynamic trapezoidal maps. *Computational Geometry: Theory and Applications*, 17:153–163, 2000.