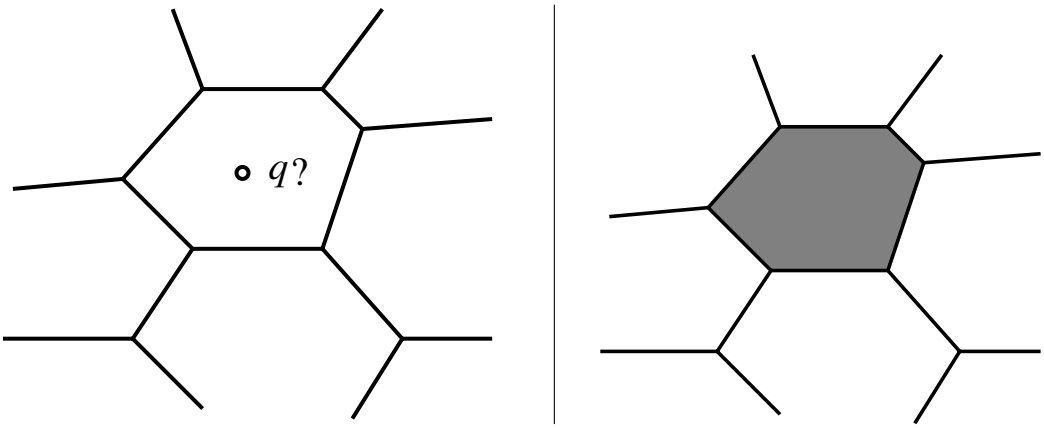

INPUT OUTPUT

17.7 Point Location

Input description: A decomposition of the plane into polygonal regions and a query point q.

Problem description: Which region contains the query point q?

Discussion: Point location is a fundamental subproblem in computational geometry, usually needed as an ingredient to solve larger geometric problems. In a typical police dispatch system, the city will be partitioned into different precincts or districts. Given a map of regions and a query point (the crime scene), the system must identify which region contains the point. This is exactly the problem of planar point location. Variations include:

- *Is a given point inside or outside of polygon P?* – The simplest version of point location involves only two regions, inside-P and outside-P, and asks which contains a given query point. For polygons with lots of narrow spirals, this can be surprisingly difficult to tell by inspection. The secret to doing it both by eye or machine is to draw a ray starting from the query point and ending beyond the furthest extent of the polygon. Count the number of times the polygon crosses through an edge. The query point will lie within the polygon iff this number is odd. The case of the line passing through a vertex instead of an edge is evident from context, since we are counting the number of times we pass through the boundary of the polygon. Testing each of the n edges for intersection against the query ray takes $O(n)$ time. Faster

algorithms for convex polygons are based on binary search, and take $O(\lg n)$ time.

- *How many queries must be performed?* – It is always possible to perform this inside-polygon test separately on each region in a given planar subdivision. However, it would be wasteful to perform many such point location queries on the same subdivision. It would be much better to construct a grid-like or tree-like data structure on top of our subdivision to get us near the correct region quickly. Such search structures are discussed in more detail below.

- *How complicated are the regions of your subdivision?* – More sophisticated inside-outside tests are required when the regions of your subdivision are arbitrary polygons. By triangulating all polygonal regions first, each inside-outside test reduces to testing whether a point is in a triangle. Such tests can be made particularly fast and simple, at the minor cost of recording the full polygon name for each triangle. An added benefit is that the smaller your regions are, the better grid-like or tree-like superstructures are likely to perform. Some care should be taken when you triangulate to avoid long skinny triangles, as discussed in Section 17.3 (page 572).

- *How regularly sized and spaced are your regions?* – If all resulting triangles are about the same size and shape, the simplest point location method imposes a regularly-spaced $k \times k$ grid of horizontal and vertical lines over the entire subdivision. For each of the k^2 rectangular regions, we maintain a list of all the regions that are at least partially contained within the rectangle. Performing a point location query in such a *grid file* involves a binary search or hash table lookup to identify which rectangle contains query point q, and then searching each region in the resulting list to identify the right one.

  Such grid files can perform very well, provided that each triangular region overlaps only a few rectangles (thus minimizing storage space) and each rectangle overlaps only a few triangles (thus minimizing search time). Whether it performs well depends on the regularity of your subdivision. Some flexibility can be achieved by spacing the horizontal lines irregularly, depending upon where the regions actually lie. The *slab method*, discussed below, is a variation on this idea that guarantees worst-case efficient point location at the cost of quadratic space.

- *How many dimensions will you be working in?* – In three or more dimensions, some flavor of kd-tree will almost certainly be the point-location method of choice. They may also be the right answer for planar subdivisions that are too irregular for grid files.

  Kd-trees, described in Section 12.6 (page 389), decompose the space into a hierarchy of rectangular boxes. At each node in the tree, the current box is split into a small number (typically 2, 4, or 2^d for dimension d) of smaller boxes. Each leaf box is labeled with the (small) set regions that are at least

partially contained in the box. The point location search starts at the root of the tree and keeps traversing down the child whose box contains the query point q. When the search hits a leaf, we test each of the relevant regions to see which one of them contains q. As with grid files, we hope that each leaf contains a small number of regions and that each region does not cut across too many leaf cells.

- *Am I close to the right cell?* – Walking is a simple point-location technique that might even work well beyond two dimensions. Start from an arbitrary point p in an arbitrary cell, hopefully close to the query point q. Construct the line (or ray) from p to q and identify which face of the cell this hits (a so-called *ray shooting query*). Such queries take constant time in triangulated arrangements.

  Proceeding to the neighboring cell through this face gets us one step closer to the target. The expected path length will be $O(n^{1/d})$ for sufficiently regular d-dimensional arrangements, although linear in the worst case.

The simplest algorithm to guarantee $O(\lg n)$ worst-case access is the *slab* method, which draws horizontal lines through each vertex, thus creating $n + 1$ "slabs" between the lines. Since the slabs are defined by horizontal lines, finding the slab containing a particular query point can be done using a binary search on the y-coordinate of q. Since there can be no vertices within any slab, the region containing a point within a slab can be identified by a second binary search on the edges that cross the slab. The catch is that a binary search tree must be maintained for each slab, for a worst-case of $O(n^2)$ space. A more space-efficient approach based on building a hierarchy of triangulations over the regions also achieves $O(\lg n)$ for search and is discussed in the notes below.

Worst-case efficient computational geometry methods either require a lot of storage or are fairly complicated to implement. We identify implementations of worst-case methods below, which are worth at least experimenting with. However, we recommend kd-trees for most general point-location applications.

Implementations: Both CGAL (*www.cgal.org*) and LEDA (see Section 19.1.1 (page 658)) provide excellent support for maintaining planar subdivisions in C++. CGAL favors a jump-and-walk strategy, although a worst-case logarithmic search is also provided. LEDA implements expected $O(\lg n)$ point location using partially persistent search trees.

ANN is a C++ library for both exact and approximate nearest-neighbor searching in arbitrarily high dimensions. It can be used to quickly identify a nearby cell boundary point to begin walking from. Check it out at *http://www.cs.umd.edu/~mount/ANN/*.

Arrange is a package for maintaining arrangements of polygons in either the plane or on the sphere. Polygons may be degenerate, and hence represent arrangements of lines. A randomized incremental construction algorithm is used, and efficient point location on the arrangement is supported.

Arrange is written in C by Michael Goldwasser and is available from *http://euler.slu.edu/∼goldwasser/publications/*.

Routines (in C) to test whether a point lies in a simple polygon have been provided by [O'R01, SR03].

Notes: Snoeyink [Sno04] gives an excellent survey of the state-of-the-art in point location, both theoretical and practical. Very thorough treatments of deterministic planar-point location data structures are provided by [dBvKOS00, PS85].

Tamassia and Vismara [TV01] use planar point location as a case study of geometric algorithm engineering, in Java. An experimental study of algorithms for planar point location is described in [EKA84]. The winner was a bucketing technique akin to the grid file.

The elegant triangle refinement method of Kirkpatrick [Kir83] builds a hierarchy of triangulations above the actual planar subdivision such that each triangle on a given level intersects only a constant number of triangles on the following level. Since each triangulation is a fraction of the size of the subsequent one, the total space is obtained by summing up a geometric series and hence is linear. Furthermore, the height of the hierarchy is $O(\lg n)$, ensuring fast query times. An alternative algorithm realizing the same time bounds is [EGS86]. The slab method described above is due to [DL76] and is presented in [PS85]. Expositions on the inside-outside test for simple polygons include [Hai94, O'R01, PS85, SR03].

More recently, there has been interest in dynamic data structures for point location, which support fast incremental updates of the planar subdivision (such as insertions and deletions of edges and vertices) as well as fast point location. Chiang and Tamassia's [CT92] survey is an appropriate place to begin, with updated references in [Sno04].

Related Problems: Kd-trees (see page 389), Voronoi diagrams (see page 576), nearest neighbor search (see page 580).