The Sweepline Algorithm for All Nearest Neighbors

Rex A. Dwyer

One important and expanding subfield of computer science is *computational geometry*. This area deals with algorithms and data structures for efficiently solving problems involving geometric objects like points, lines, rectangles and other polygons, circles, and spheres. Important geometric problem arise in software systems for computer-aided design (CAD) for laying out printed and integrated circuits, robotics, statistics, and other applications. One appealing feature of computational geometry is that many of its problems are very easy to state, yet cannot be solved efficiently without sophisticated techniques.

One such problem is the *All-Nearest-Neighbors Problem*. A set of input points is given in the form of a list of x- and y-coordinates. To avoid confusion, we will call the input points "sites", and reserve "point" to refer to other points. The task is to find, for each site, its nearest neighbor among the remaining sites. Fig. 1 shows a set of sites, each with an arrow pointing to its nearest neighbor. It is easy to solve this problem in $\Theta(n^2)$ time, but it is actually possible to solve it in $\Theta(n \log n)$ time. Several algorithms are this fast; we will study one, a *sweepline* algorithms.

First of all, we will divide the work into three phases. The first phase will find the *upper* nearest neighbor of each site; this is the nearest site among those with greater y-coordinate. The second phase will find all *lower* nearest neighbors. The third phase will compare the upper and lower nearest neighbors of each site to find the *overall* nearest neighbor. Upper and lower nearest neighbors are shown in Figs. 2 and 3. We will concentrate on the first phase, since the second phase is similar, and the third phase is very easy.

Conceptually, the first phase involves a horizontal line — the *sweepline* — advancing slowly downward from the highest site to the lowest. The upper nearest neighbor of a site will be determined at the moment the sweepline touches the site in its downward path. Let us jump ahead and assume that the first phase is well underway as shown in Fig. 4, where the sweepline is pausing between two sites. Consider the (infinitely many) points lying on the sweepline. For each, which site above the sweepline is nearest? It is fairly clear that not every site is the nearest neighbor of a point on the sweepline; those that are, are called *active* sites. A site becomes active when the sweepline passes through it. It remains active for a while, then becomes

permanently inactive. In Fig. 4, black dots mark active sites, and hollow or gray dots mark inactive sites.

The most important data structure in the algorithm is a dictionary of the active sites. Each active site is the nearest neighbor of a single segment (or ray) of the sweepline. The boundaries of these segments are defined by bisectors of pairs of active sites. The order of the segments along the sweepline is the same as the order of the corresponding sites when sorted by x-coordinate. Therefore, it is most useful to organize the dictionary according to x-coordinates. For the moment, let us assume that the dictionary is implemented by a double-linked list with one record per site, each having a pointer to the site's successor (next larger x) and another to its predecessor (next smaller x). (Later, we will replace this with a more efficient implementation.)

What happens when the sweepline is advanced slightly? The boundaries of the segments shift to the left and right, and, more importantly, the segments themselves shrink or grow. If a site's segment is growing, the site is a *waxer*; if shrinking, a *waner*. The leftmost and rightmost sites always have semi-infinite segments, but we will call them waxers. When a site becomes active, it is a waxer; subsequently, it may become a waner, and then become inactive. Waners never become waxers. The lifecycle of a site is (inactive→waxer→waner→inactive). (Some sites are "immortal" and never become waners.)

Whether an active site is a waxer or a waner depends only on its predecessor and successor on the sweepline. To determine quickly whether a site is a waner or a waxer, we can look at the triangle formed by the three sites. If the sequence (site→predecessor→successor→site) makes a clockwise tour around this triangle, the site is a waxer; if counterclockwise, a waner. Mathematically, we can determine this by evaluating the determinant

$$\begin{vmatrix} x & y & 1 \\ x_{\text{pred}} & y_{\text{pred}} & 1 \\ x_{\text{succ}} & y_{\text{succ}} & 1 \end{vmatrix},$$

where (x, y) is the site, $(x_{\text{pred}}, y_{\text{pred}})$ is its predecessor, and $(x_{\text{succ}}, y_{\text{succ}})$ is its successor. The absolute value of this determinant is two times the area of the triangle, which is of no interest to us. The *sign*, however, tells whether the vertices are listed in clockwise or counterclockwise order.

When we advance the sweepline through the sites, we will only be interested in stopping it at the y-values where the set of active sites changes. These stopping points are the *events*, and we will simply jump from event to event. There are two kinds of events: *insertion events* occur at the y-value of every site, and *deletion events* occur at the y-value where a waner's segment disappears and the site becomes inactive. The y-values for all the insertion events are known at the beginning of the program, but the y-values of the deletion events are learned only as the program progresses. Events are stored in a priority queue prioritized on y-values. At each step, the next event is determined by performing a DELETEMAX, so that events are performed from top to bottom.

Since a waner will eventually disappear from the sweepline even if no new sites are encountered, a deletion event must be present in the event queue for every active waner. The y-value for this deletion depends only on its predecessor and successor. In fact, it is the y-coordinate of the *center of the circle* defined by the three sites. Whenever any site is deleted from or inserted into the active dictionary, its predecessor's successor and successor's predecessor change. So it is necessary to reexamine these neighbors to see whether they have become waners for which deletions must be scheduled.

Deletion events are the easiest to carry out. If still active, the site is deleted. Its former predecessor and successor are examined and, if necessary, new deletions events are scheduled for one or both.

Insertion events are more complicated. The new site is obviously becoming active, so it must be inserted into the active dictionary according to its x-coordinate. Its predecessor now has a new successor. Is the predecessor still a waxer? If so, it remains in place and no deletion event needs to be scheduled. If not, either the predecessor must be deleted immediately, or a future deletion must be scheduled. It must be deleted immediately if the center of its circle lies above the sweepline. Otherwise, a future deletion event must be scheduled. If it is deleted immediately, *its* predecessor has a new neighbor and the process must be repeated. This continues until the new site's predecessor is found to be one that must remain active. A similar process occurs with the new site's successor. It may be helpful to imagine the new site conquering territory to its left and then to its right. On each side, some sites lose all their territory, one loses some but not all, and others further from the new site are unaffected.

In Fig. 4, the next event will be the insertion of the gray site just below

the sweepline on the left. It will be inserted between the third and fourth active sites. Next, the third site will be considered for deletion. Since it is definitely a waner after the insertion, the center of the circle passing through the second, third and new sites is computed. Since it lies above the sweepline, the third site is deleted. Next, the second site is considered. It is a waner, and the center of the circle defined by the second, first, and new sites is above the sweepline, so it, too, is deleted. The first site is leftmost is not deleted, so the process ends on the left side of the new site. On the right side, two sites are deleted, leaving the configuration of Fig. 5. The sixth site has become a waner, so a deletion event must be scheduled. The y-value for the event is found by extending the bisectors on either side of the site to their intersection point. This deletion event pops up a few steps later, as shown in Fig. 6. Fig. 7 shows the result of the deletion event. This deletion causes the third active site in Fig. 7 to become a waner, so another deletion event must be scheduled.

We have developed a method for keeping track of the active sites, but we have not determined how to find the nearest neighbor! In fact, the nearest neighbor of a site is always examined during its insertion event, so it is only necessary to compute distances to sites as we consider whether or not to delete them.

Analysis of Running Time

Suppose there are n sites. First we will figure out how many insertions and deletions occur, and how many insertion and deletion events are scheduled. You may have noticed that many deletion events can be scheduled for a single site; only the first one actually performs a deletion. On the other hand, many deletions are performed by insertion events rather than deletion events. So it is important to distinguish between *deletions* and *deletion events*.

It is clear that there are n insertions, at most n deletions, and n insertion events. Counting deletion *events* is more difficult. Some deletion events are scheduled by insertion events. No single insertion event can schedule more than two deletion events, so at most $2n$ deletion events are scheduled this way.

Some deletion events are scheduled by other deletion events, but only if the deletion event *really deletes*. Since there are at most n deletions, at most

$2n$ deletion events are scheduled in this way. The total number of deletion events is at most $4n$, and the total number of events is at most $5n$.

Considering the priority queue, we see that there are $5n$ INSERTs and $5n$ DELETEMAXs. If a leftist heap or skew heap is used to implement the priority queue, $O((5n)\log(5n)) = O(n \log n)$ time is required for priority queue operations.

Next consider the number of times the clockwise test is used, and the number of times the center of a circle is computed. Each of these requires $O(1)$ time. With a little study, you will see that each is executed at most $6n$ times, or you may be able to improve this to $4n$ times; either way, the total work is $O(n)$.

Now consider the dictionary of active sites. Insertion and deletion in a double-linked list take $O(n)$ time. A red-black, AVL, 2-3, or splay tree is a better choice. The $2n$ insertions and deletions will require $O(n \log n)$ time. Of course, finding successors and predecessors is not a constant time operation in these data structures. As an exercise, you should analyze the total time required for finding successors and predecessors; it is $O(n \log n)$.

The total running time is $O(n \log n)$.

Exercises

1. Is it possible for two or more deletion events to be scheduled for the same site?

2. Is it possible to look at a set of points and quickly determine which sites are "immortal"? Can you write an algorithm to find the immortal sites without using a sweepline?