An all-round sweep algorithm for 2-dimensional nearest-neighbor problems

Klaus Hinrichs*, Jurg Nievergelt**, Peter Schorn**
Universität Siegen* and Informatik, ETH, CH - 8092 Zürich, Switzerland**

We present a simple, efficient, robust plane-sweep algorithm that solves 2-dimensional nearest-neighbor problems in asymptotically optimal time O(n log n). A "foolproof" implementation guarantees an exact result at the cost of using triple-precision integer arithmetic at some key steps.

1 Algorithms for nearest-neighbor problem

A number of common geometric problems involve finding a nearest neighbor for all or some points in a set of given points, for example: *closest-pair*, *bichromatic closest-pair* (or equivalently, distance between two point sets), and *all-nearest-neighbors*. We consider the 2-dimensional versions of these related problems, and study in detail the *all-nearest-neighbors problem*: Given a set S of n points in the plane, find a nearest neighbor of each with respect to the Euclidean metric. Other problems are solved by simple modifications of the all-nearest-neighbors algorithm.

Known algorithms with optimal worst case time complexity O(n log n) are of two types:
1) Extract the solution (in linear time) from the answer to the more general problem of constructing the Voronoi diagram of the given points, computed in time O(n log n) [Fo 87].
2) Compute the desired result directly, without the additional information present in the Voronoi diagram. [Va 89] describes an algorithm which works for any number of dimensions in any L_p-metric. It maintains a growing set of shrinking boxes; when a box has shrunk to a single point, its associated information determines a nearest neighbor.

We show how the 2-dimensional all-nearest-neighbors problem is solved by an asymptotically optimal sweep algorithm hat is efficient and robust in practice.

Sweep algorithms impose a left-to-right order on the data. This makes it easier to solve the two problems *all-nearest-neighbors-to-the-left* and *all-nearest-neighbors-to-the-right,* and to combine their solutions in time O(n). In [HNS 89] we presented the first sweep algorithm for this problem, based on the following invariant: 'The cross-section is the intersection of the sweep-line with the Voronoi-diagram of the points to the left of the sweep-line'. This implies that the y-table stores formulas of bisectors to be evaluated for different values of x. Continuing attempts to simplify and improve the algorithm's robustness in the presence of round-off errors have led to what we believe is the simplest and most robust sweep algorithm for nearest-neighbors problem: Instead of bisectors, the y-table maintains a set of 'active points', ordered by their original y-coordinates, thus guaranteeing that the points are maintained in correct vertical order. Events in the x-queue are of two types: Data points correctly ordered by their x-coordinates, and deactivation events; the latter are computed as intersections of bisectors and may thus be afflicted by round-off errors. The use of triple-precision arithmetic at some crucial steps removes any error from the entire computation.

2 Geometric preliminaries

2.1 Analytic geometry of points, lines, and circles

E^2 : the Euclidean plane; $u = (u_x, u_y)$ and $v = (v_x, v_y) \in E^2$; d(u, v): Euclidian distance
For 3 points u, v and w define the determinant

$$D(u,v,w) = \begin{vmatrix} u_x & u_y & 1 \\ v_x & v_y & 1 \\ w_x & w_y & 1 \end{vmatrix} = (u_x - v_x)(v_y - w_y) - (u_y - v_y)(v_x - w_x)$$

collinear(u, v, w) $\Leftrightarrow$ D(u, v, w) = 0; rightturn(u, v, w) $\Leftrightarrow$ D(u, v, w) < 0

Lemma 1(bounded determinant):
If 3 points u, v, w are contained in the rectangle given by its opposite corners (x_{min}, y_{min}) and (x_{max}, y_{max}), then $|D(u, v, w)| \le (x_{max} - x_{min})(y_{max} - y_{min})$.

2.2 Data points and their classification

S: a finite set of points in E^2, called data points. s, q, u, v, w, .. $\in$ S.
L: the sweep line, a vertical line at x-coordinate x_L.

p: a point on L; p may be a data point $\in$ S, or p $\notin$ S.
S_{left} : the set of those points of S that lie to the left of L or on L.

$s \in S_{left}$ is *active* $\Leftrightarrow \exists p \in L \; \forall q \in S_{left}$: $d(s, p,) \le d(q, p)$
In words, a point in S_{left} is active iff it is a nearest neighbor to the left of some point p on L.

S_{active} (L): the set of all active points. Notice: $S_{active} \subseteq S_{left} \subseteq S$.
y-order $<_y$ on S_{active} : for u, v $\in S_{active}$, define $u <_y v :\Leftrightarrow u_y < v_y$

Lemma 2 ($<_y$ is a total order): No two points in S_{active} have equal y-coordinates.

predecessor and successor function w.r.t $<_y$: pred(), succ(): $S_{active} \rightarrow S_{active} \cup \{nil\}$ (A sentinel)
u, v $\in S_{active}$ form a consecutive pair $\Leftrightarrow$ u = pred(v); $bs_{u,v}$ is the bisector of u and v.
u, v, w $\in S_{active}$ form a consecutive triple $\Leftrightarrow$ u = pred(v), succ(v) = w

Lemma 3 (monotonically increasing distances):
Let $p \in L$, let u, v $\in S_{active}$ with $v_y < u_y < p_y$ or $p_y < u_y < v_y$. Then $d(u, p) < d(v, p)$.

For a consecutive triple (u, v, w) with center(u, v, w) to the right of L, i.e. $center(u, v, w)_x > x_L$, center(u, v, w) is called the 'deactivation event of v'.

Lemma 4 (local activity test):
For a consecutive triple (u, v, w) $\in S_{active}$: rightturn(u, v, w) $\Rightarrow center(u, v, w)_x > x_L$

Lemma 6 (bisectors partition the sweep line):
The bisectors $bs_{u,v}$ for all consecutive pairs u, v $\in S_{active}$ partition the sweep line L into pairwise disjoint open intervals. For each $s \in S_{active}$ there is exactly one interval I(s) such that

$\forall p \in I(s) \; \forall q \in S_{left}, q \ne s$: $d(p, s) < d(p, q)$

Lemma 7 (updating S_{active} at a deactivation event):
Let L and L' be consecutive stops of the sweep line: between L and L' there are no data points and no deactivations, but L' contains the deactivation event of v. Then $S_{active}(L') = S_{active}(L) - \{v\}$

Lemma 8 (updating S_{active} at a data point):
Let L and L' be consecutive positions of the sweep line, but L' contains the data point p. Then $S_{active}(L') = S_{active}(L) \cup \{p\} - D_{up} - D_{down}$ where:
$D_{up} = \{q \in S_{active} / q_y \ge p_y$, succ(q) $\in S_{active} \Rightarrow$ (p, q, succ(q)) fails the local activity test $\}$, and
$D_{down} = \{q \in S_{active} / q_y \le p_y$, pred(q) $\in S_{active} \Rightarrow$ (pred(q), q, p) fails the local activity test $\}$.

3. The sweep invariant: S_{active} (L) stored in the y-table, to be updated in the following cases:

3.1 Deactivation event: The sweep line L reaches the center of the circle of the consecutive triple (u, v, w) with rightturn(u, v, w). This is called an deactivation event for v (lemma 7).

A deactivation event for v created by the consecutive triple (u, v, w) is processed by removing v from the y-table, then testing whether the consecutive triples (pred(u), u, w) and (u, w, succ(w)) generate new deactivation events to be inserted into the x-queue.

3.2 Data point: A new data point p ∈ S is inserted into the y-table. We have to test whether the consecutive triples (p, succ(p), succ(succ(p))) and (pred(pred(p)), pred(p), p), if they exist, pass the local activity test.

3.3 Finding the nearest neighbor to the left

After updating the invariant the nearest neighbor to the left of a data point p is found among pred(p), succ(p) and the points which were removed by p from the y-table. Lemma 3 guarantees that the remaining points in the y-table have a larger distance from p than succ(p) or pred(p). While updating the invariant we maintain the nearest neighbor to the left of p found so far.

4. Program structure

```
procedure transition(p); { Incorporates all the work to be done in processing an event.}
begin
   if dataPoint(p) then
         insertY(p);
         nn := nil;
         updateY(p, nn, succY);
         updateY(p, nn, predY)
   elsif deactivate(p) then
         u := succY(p);
         v := predY(p);
         deleteY(p);
         testForDeactivate(predY(v), v, u);
         testForDeactivate(v, u, succY(u));
   end { testForDeactivate(u,v,w) schedules the leftmost deactivation for v, if necessary }
end transition;

procedure updateY(p, nn, nextY);
begin
   u := nextY(p);
   v := nextY(u);
   while (v ≠ nil) cand rightturn(p, u, v)
         cand (centerx(p, u, v) < $p_x$) do
         if d(p, u) < d(p, nn) then nn := u; end;
         deleteY(u);
         u := v;
         v := nextY(u);
   end;
   if d(p, u) < d(p, nn) then nn := u; end;
   testForDeactivate(p, u, v);
end updateY;
```

5. Analysis: Time O(n log n) by conventional techniques.

6. An exact implementation: We have implemented this algorithm in such a way that it handles all degenerate configurations and achieves an exact result. The degeneracies to be dealt with are events of equal x-coordinate, including intersection events that coincide (equal x- and y-coordinates). If input coordinates are integers, triple precision integer arithmetic guarantees exact results.

6.1 Disambiguating degenerate configurations: by extending the order on events in the x-queue and points in the y-table to the case of equal x- and/or y-coordinates. Deactivation events with (near-)equal x-coordinates can be processed in any order - we need not break the tie in this case.

6.2 Bounding the computed quantities

Assuming all data points (x, y) lie on a finite integer grid, $|x| < M$ and $|y| < M$, the evaluation of:

$$\text{center}(u,v,w)_x = \frac{1}{2D(u,v,w)} \begin{vmatrix} u_x^2 + u_y^2 & u_y & 1 \\ v_x^2 + v_y^2 & v_y & 1 \\ w_x^2 + w_y^2 & w_y & 1 \end{vmatrix}$$

requires no numbers larger than $4\ M^3$ (Lemma 1). Evaluating this quantity using truncated division "div" instead of real division "/" may cause a round-off error that has provably no effect on the algorithm, due to the following facts: 1) Data points lie on grid points only, 2) The absolute error is < 1, and 3) Two consecutive deactivation events that are not separated by any data point can be processed in any order.
Conclusion: Let m be the number of bits used to represent the input data. Integer arithmetic with 3m +2 bits yields exact results. Note also that this is a considerable improvement over the straightforward rational arithmetic approach which requires five-fold precision.

7. Performance and optimization

We have implemented this sweep algorithm in an object oriented extension of Pascal on the Apple Macintosh II computer. The running times for 10^2, 10^3 and 10^4 random points uniformly distributed in a square are 0.6, 8.7 and 130.4 seconds. To show that this direct solution compares favorably with 'detours' via the Voronoi diagram, we implemented the most efficient algorithm for the Voronoi diagram, Fortune's sweep [Fo 87], on the same system. The Voronoi computation on the same point sets took at least three times longer, namely 2.3, 28 and 453.3 seconds.

Assuming a formula of the form $c_i * n * \log_2 n$ for the running time of both algorithms our experimental data suggests a constant $c_1 \approx 0.95$ ms for our algorithm versus $c_2 \approx 3.3$ ms for computing the Voronoi diagram. In addition, the sweep uses memory more efficiently, since the objects stored in the the x-queue and y-table are less complex than the ones needed for the Voronoi diagram. In a 4MB partition we can compute all-nearest-neighbors for up to $3.15*10^4$ points, whereas the largest possible configuration for a Voronoi diagram has about $1.3*10^4$ points.

We use the following optimizations to further improve efficiency.

- We implement the x-queue as a combination of a sorted array of all data points, and a list of current deactivation events. Implementing this list as a balanced treen guarantees O(n log n) time, but our experience shows that the number of current deactivation events is usually small, and that a linear list implementation is feasible. In test runs on configurations of 10^3 and 10^4 uniformly distributed random points the x-queue never held more than 20 and 60 deactivation events, respectively
- Deactivation events to the right of S are discarded.
- A simple geometric observation shows that a triple (p, u, v) where p on L, $u_y \geq p_y$, (u, v) a consecutive pair, fails the local activity test if d(p, v) < d(p, u). Since these comparisons have to be made anyway in the course of finding the nearest neighbor, we can sometimes save the more expensive computation of the center of a circle. A similar criterion holds if (u, v) lies below p.

8. Conclusions

The *all-nearest-neighbors* algorithm presented is easily adapted to solving the bichromatic-closest-pair (distance between two point sets) problem: Given two sets P, Q of points, identify a pair $p \in P$, $q \in Q$ that minimizes the distance d(p, q). The points of P (only) are entered in the y-table, the points of Q serve as query events.

The two major issues that decide whether a geometric program is practical are: 1) Can it handle degenerate configurations? and 2) is it numerically robust? Few geometric algorithms have been studied thoroughly from this point of view. Our implementation addresses both problems. 1) Degenerate cases are classified into two categories: Those whose order is irrelevant, as in the case of consecutive deactivation events, and those that must be ordered consistently. 2) Assuming that the points lie on an integer grid, we guarantee numerical robustness by using triple precision integer arithmetic for computing deactivation events - a small price to pay for exact results.

References

[Fo 87] S. Fortune: A Sweepline Algorithm for Voronoi Diagrams, Algorithmica 2(1987), 153-174.

[HNS 89] K. Hinrichs, J. Nievergelt, P. Schorn: A sweep algorithm and its implementation: The all-nearest-neighbors problem revisited, 14-th International Workshop on Graph-Theoretic Concepts in Computer Science (WG '88), Amsterdam, J. van Leeuwen (ed.), LNCS 344, Springer, 1989.

[Va 89] P. Vaidya: An O(n log n) Algorithm for the All-Nearest-Neighbors Problem, Discrete & Computational Geometry, 4 (1989), 101-115.