26 The closest pair problem

Sweep algorithms solve many kinds of proximity problems efficiently. We present a simple sweep that solves the two-dimensional closest pair problem elegantly in asymptotically optimal time. We explain why sweeping generalizes easily, but not efficiently, to multidimensional closest pair problems.

26.1 The problem

We consider the two-dimensional *closest pair problem*: Given a set S of n points in the plane find a pair of points whose distance δ is smallest (Fig. 26.1). We measure distance using the metric d_k, for any $k \geq 1$, or d_∞, defined as

$$d_k((x',y'),(x'',y'')) = \sqrt[k]{|x'-x''|^k + |y'-y''|^k}$$
$$d_\infty((x',y'),(x'',y'')) = \max(|x'-x''|,|y'-y''|)$$

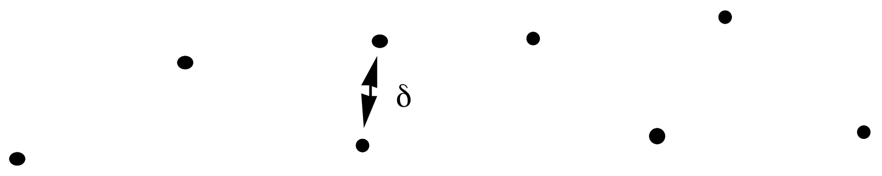

Figure 26.1: Identify a closest pair among n points in the plane.

Special cases of interest include the "Manhattan metric" d_1, the "Euclidean metric" d_2, and the "maximum metric" d_∞. Figure 26.2 shows the "circles" of radius 1 centered at a point p for some of these metrics.

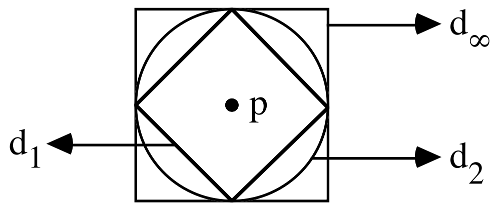

Figure 26.2: The results of this chapter remain valid when distances are measured in various metrics.

The closest pair problem has a lower bound $\Omega(n \cdot \log n)$ in the algebraic decision tree model of computation [PS 85]. Its solution can be obtained in asymptotically optimal time $O(n \cdot \log n)$ as a special case of more general problems, such as 'all-nearest-neighbors' [HNS 92] (for each point, find a nearest neighbor), or constructing the

Voronoi diagram [SH 75]. These general approaches call on powerful techniques that make the resulting algorithms harder to understand than one would expect for a simply stated problem such as "find a closest pair". The divide-and-conquer algorithm presented in [BS 76] solves the closest pair problem directly in optimal worst-case time complexity $\Theta(n \cdot \log n)$ using the Euclidean metric d_2. Whereas the recursive divide-and-conquer algorithm involves an intricate argument for combining the solutions of two equally sized subsets, the iterative plane-sweep algorithm [HNS 88] uses a simple incremental update: Starting with the empty set of points, keep adding a single point until the final solution for the entire set is obtained. A similar plane-sweep algorithm solves the closest pair problem for a set of convex objects [BH 92].

26.2 Plane-sweep applied to the closest pair problem

The skeleton of the general sweep algorithm presented in section 25.2, with the data structures x-queue and y-table, is adapted to the closest pair problem as shown in Fig. 26.3. The *x-queue* stores the points of the set S, ordered by their x-coordinate, as events to be processed when updating the vertical cross section. Two pointers into the x-queue, 'tail' and 'current', partition S into four disjoint subsets:

1. The discarded points to the left of 'tail' are not accessed any longer.
2. The active points between 'tail' (inclusive) and 'current' (exclusive) are being queried.
3. The current transition point, p, is being processed.
4. The future points have not yet been looked at.

The y-table stores the active points only, ordered by their y-coordinate.

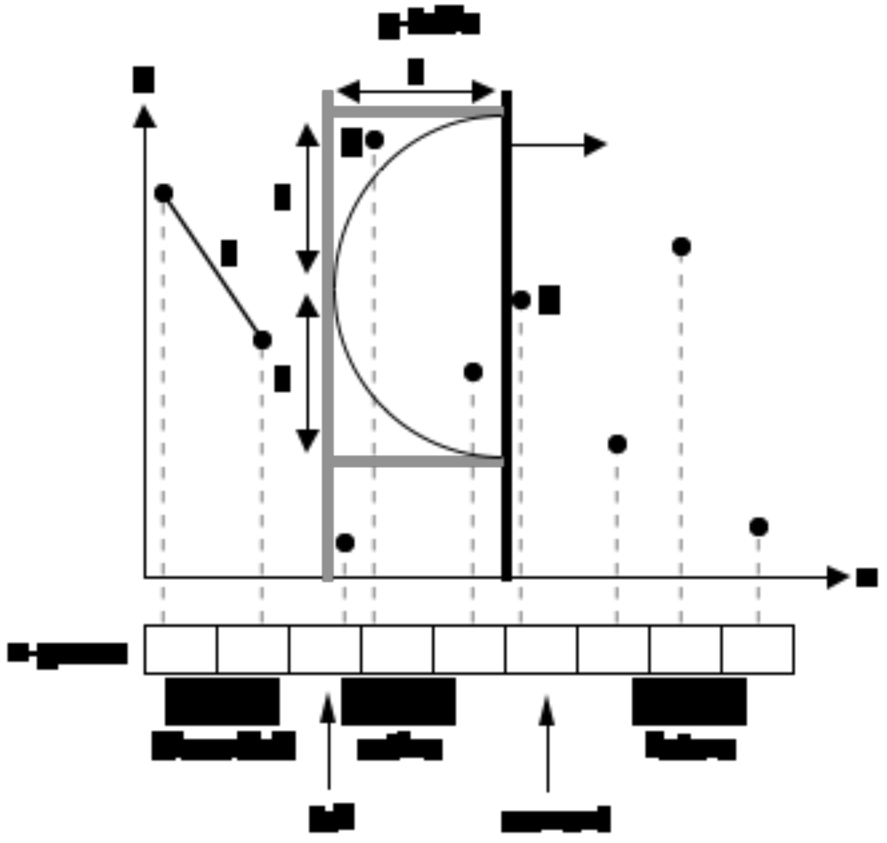

Figure 26.3: Updating the invariant as the next point p is processed.

We need to compare points by their x-coordinates when building the x-queue, and by their y-coordinates while sweeping. For simplicity of presentation we assume that no two points have equal x- or y-coordinates. Points with equal x- or y-coordinates are handled by imposing an arbitrary, but consistent, total order on the set of points. We achieve this by defining two lexicographic orders: $<_x$ to be used for the x-queue, $<_y$ for the y-table:

$$(x', y') <_x (x'', y'') \;:\Leftrightarrow\; (x' < x'') \vee ((x' = x'') \wedge (y' < y''))$$

$$(x', y') <_y (x'', y'') \;:\Leftrightarrow\; (y' < y'') \vee ((y' = y'') \wedge (x' < x'')).$$

The program of the following section initializes the x-queue and y-table with the two leftmost points being active, with δ equal to their distance, and starts the sweep with the third point.

The distinction between discarded and active points is motivated by the following argument. When a new point p is encountered we wish to answer the question whether this point forms a closest pair with one of the points to its left. We keep a pair of closest points seen so far, along with the corresponding minimal distance δ. Therefore, all candidates that may form a new closest pair with the point p on the sweep line lie in a half circle centered at p, with radius δ.

The key question to be answered in striving for efficiency is how to retrieve quickly all the points seen so far that lie inside this half circle to the left of p, in order to compare their distance to p against the minimal distance δ seen so far. We may use any helpful data structure that organizes the points seen so far, as long as we can update this data structure efficiently across a transition. A circle (or half-circle) query is complex, at least when embedded in a plane-sweep algorithm that organizes data according to an orthogonal coordinate system. A rectangle query can be answered more efficiently. Thus we replace the half-circle query with a bounding rectangle query, accepting the fact that we might include some extraneous points, such as q.

The rectangle query in Fig. 26.3 is implemented in two steps. First, we cut off all the points to the left at distance $\geq \delta$ from the sweep line. These points lie between 'tail' and 'current' in the x-queue and can be discarded easily by advancing 'tail' and removing them from the y-table. Second, we consider only those points q in the δ-slice whose vertical distance from p is less than δ: $|q_y - p_y| < \delta$. These points can be found in the y-table by looking at successors and predecessors starting at the y-coordinate of p. In other words, we maintain the following invariant across a transition:

1. δ is the minimal distance between a pair of points seen so far (discarded or active).
2. The active points (found in the x-queue between 'tail' and 'current', and stored in the y-table ordered by y-coordinates) are exactly those that lie in the interior of a δ-slice to the left of the sweep line.

Therefore, processing the transition point p involves three steps:
1. Delete all points q with $q_x \leq p_x - \delta$ from the y-table. They are found by advancing 'tail' to the right.
2. Insert p into the y-table.
3. Find all points q in the y-table with $|q_y - p_y| < \delta$ by looking at the successors and predecessors of p. If such a point q is found and its distance from p is smaller than δ, update δ and the closest pair found so far.

26.3 Implementation

In the following implementation the x-queue is realized by an array that contains all the points sorted by their x-coordinate. 'closestLeft' and 'closestRight' describe the pair of closest points found so far, n is the number of points under consideration, and t and c determine the positions of 'tail' and 'current':

```
xQueue: array[1 .. maxN] of point;
closestLeft, closestRight: point;
t, c, n: 1 .. maxN;
```

The x-queue is initialized by

```
procedure initX;
```

'initX' stores all the points into the x-queue, ordered by their x-coordinates.
The empty y-table is created by

```
procedure initY;
```

A new point is inserted into the y-table by

```
procedure insertY(p: point);
```

A point is deleted from the y-table by

```
procedure deleteY(p: point);
```

The successor of a point in the y-table is returned by

```
function succY(p: point): point;
```

The predecessor of a point in the y-table is returned by

```
function predY(p: point): point;
```

The initialization part of the plane-sweep is as follows:

```
initX;  initY;
closestLeft := xQueue[1];  closestRight := xQueue[2];
delta := distance(closestLeft, closestRight);
insertY(closestLeft);  insertY(closestRight);
c := 3;
```

The events are processed by the loop:

```
while c ≤ n do  begin  transition;  c := c + 1;  { next event }  end;
```

The procedure 'transition' encompasses all the work to be done for a new point:

```
procedure transition;
begin

  { step 1: remove points outside the δ-slice from the y-table }
  current := xQueue[c];
  while  current.x – xQueue[t].x ≥ delta  do  begin
    deleteY(xQueue[t]);  t := t + 1
  end;
  { step 2: insert the new point into the y-table }
  insertY(current);

  { step 3a: check the successors of the new point in the y-table }
  check := current;
  repeat
    check := succY(check);
    newDelta := distance(current, check);
    if newDelta < delta then begin
      delta := newDelta;
      closestLeft := check; closestRight := current;
    end;
  until check.y – current.y > delta;

  { step 3b: check the predecessors of the new point in the y-table }
  check := current;
  repeat
    check := predY(check);
    newDelta := distance(current, check);
    if newDelta < delta then begin
      delta := newDelta;
      closestLeft := check; closestRight := current;
    end;
  until current.y – check.y > delta;

end; { transition }
```

26.4 Analysis

We show that the algorithm described can be implemented so as to run in worst-case time $O(n \cdot \log n)$ and space $O(n)$.

If the y-table is implemented by a balanced binary tree (e.g., an AVL-tree or a 2-3-tree) the operations 'insertY', 'deleteY', 'succY', and 'predY' can be performed in time $O(\log n)$. The space required is $O(n)$.

'initX' builds the sorted x-queue in time O(n · log n) using space O(n). The procedure 'deleteY' is called at most once for each point and thus accumulates to O(n · log n). Every point is inserted once into the y-table, thus the calls of 'insertY' accumulate to O(n · log n).

There remains the problem of analyzing step 3. The loop in step 3a calls 'succY' once more than the number of points in the upper half of the bounding box. Similarly, the loop in step 3b calls 'predY' once more than the number of points in the lower half of the bounding box. A standard counting technique shows that the bounding box is sparsely populated: For any metric d_k, the box contains no more than a small, constant number c_k of points, and for any k, $c_k \leq 8$. Thus 'succY' and 'predY' are called no more than 10 times, and step 3 costs time O(log n).

The key to this counting is the fact that no two points in the y-table can be closer than δ, and thus not many of them can be packed into the bounding box with sides δ and $2 \cdot \delta$. We partition this box into the eight pairwise disjoint, mutually exhaustive regions shown in Fig. 26.4. These regions are half circles of diameter δ in the Manhattan metric d_1, and we first argue our case only when distances are measured in this metric. None of these half-circles can contain more than one point. If a half-circle contained two points at distance δ, they would have to be at opposite ends of the unique diameter of this half-circle. But these endpoints lie on the left or the right boundary of the bounding box, and these two boundary lines cannot contain any points, for the following reasons:

1. No active point can be located on the left boundary of the bounding box; such a point would have been thrown out when the δ-slice was last updated.
2. No active point can exist on the right boundary, as that x-coordinate is preempted by the transition point p being processed (remember our assumption of unequal x-coordinates).

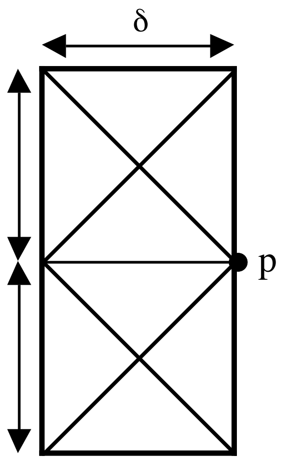

Figure 26.4: Only few points at pairwise distance ≥ δ can populate a box of size 2·δ by δ.

We have shown that the bounding box can hold no more than eight points at pairwise distance ≥ δ when using the Manhattan metric d_1. It is well known that for any points p, q, and for any k > 1: $d_1(p,q) > d_k(p,q) > d_\infty(p,q)$. Thus the bounding box can hold no more than eight points at pairwise distance ≥ δ when using any distance d_k or d_∞.

Therefore, the calculation of the predecessors and successors of a transition point costs time O(log n) and accumulates to a total of O(n · log n) for all transitions. Summing up all costs results in O(n · log n) time and O(n) space complexity for this algorithm. Since Ω(n · log n) is a lower bound for the closest pair problem, we know that this algorithm is optimal.

26.5 Sweeping in three or more dimensions

To gain insight into the power and limitation of sweep algorithms, let us explore whether the algorithm presented generalizes to higher-dimensional spaces. We illustrate our reasoning for three-dimensional space, but the same conclusion holds for any number of dimensions > 2. All of the following steps generalize easily.

Sort all the points according to their x-coordinate into the x-queue. Sweep space with a y-z plane, and in processing the current transition point p, assume that we know the closest pair among all the points to the left of p, and their distance δ. Then to determine whether p forms a new closest pair, look at all the points inside a half-sphere of radius δ centered at p, extending to the left of p. In the hope of implementing this sphere query efficiently, we enclose this half sphere in a bounding box of side length $2 \cdot \delta$ in the y- and z-dimension, and δ in the x-dimension. Inside this box there can be at most a small, constant number c_k of points at pairwise distance $\geq \delta$ when using any distance d_k or d_∞.

We implement this box query in two steps: (1) by cutting off all the points farther to the left of p than δ, which is done by advancing 'tail' in the x-queue, and (2) by performing a square query among the points currently in the y-z-table (which all lie in the δ-slice to the left of the sweep plane), as shown in Fig. 26.5. Now we have reached the only place where the three-dimensional algorithm differs substantially. In the two-dimensional case, the corresponding one-dimensional interval query can be implemented efficiently in time O(log n) using find, predecessor, and successor operations on a balanced tree, and using the knowledge that the size of the answer set is bounded by a constant. In the three-dimensional case, the corresponding two-dimensional orthogonal range query cannot in general be answered in time O(log n) (per retrieved point) using any of the known data structures. Straightforward search requires time O(n), resulting in an overall time $O(n^2)$ for the space sweep. This is not an interesting result for a problem that admits the trivial $O(n^2)$ algorithm of comparing every pair.

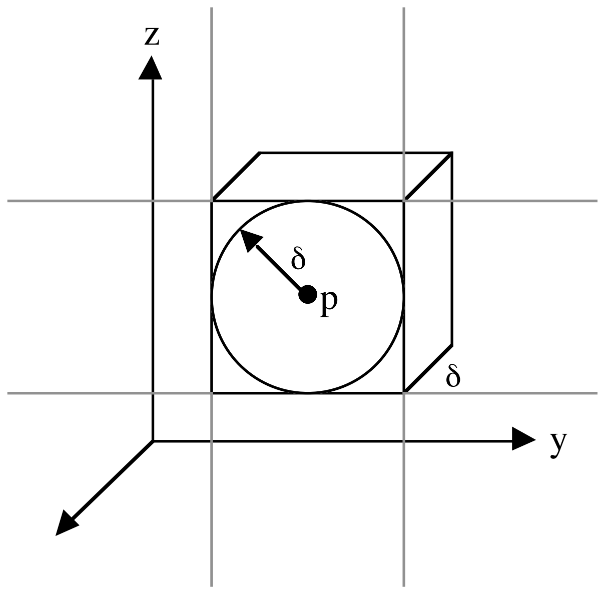

Figure 26.5: Sweeping a plane across three-dimensional space.
Ideas generalize, but efficiency does not.

Sweeping reduces the dimensionality of a geometric problem by one, by replacing one space dimension by a "time dimension". Reducing a two-dimensional problem to a sequence of one-dimensional problems is often efficient because the total order defined in one dimension allows logarithmic search times. In contrast, reducing a three-dimensional problem to a sequence of two-dimensional problems rarely results in a gain in efficiency.

Exercises

1. Consider the following modification of the plane-sweep algorithm for solving the closest pair problem [BH 92]. When encountering a transition point p do not process all points q in the y-table with $|q_y - p_y| < \delta$, but test only whether the distance of p to its successor or predecessor in the y-table is smaller than δ. When deleting a point q with $q_x \le p_x - \delta$ from the y-table test whether the successor and predecessor of q in the y-table are closer than δ. If a pair of points with a smaller distance than the current δ is found update δ and the closest pair found so far. Prove that this modified algorithm finds a closest pair What is the time complexity of this algorithm?

2. Design a divide-and-conquer algorithm which solves the closest pair problem. What is the time complexity of your algorithm? *Hint:* Partition the set of n points by a vertical line into two subsets of approximately $n/2$ points. Solve the closest pair problem recursively for both subsets. In the conquer step you should use the fact that δ is the smallest distance between any pair of points both belonging to the same subset. A point from the left subset can only have a distance smaller than δ to a point in the right subset if both points lie in a $2\cdot\delta$-slice to the left and to the right of the partitioning line. Therefore, you only have to match points lying in the left δ-slice against points lying in the right δ-slice.