The closest pair problem: a randomized incremental algorithm

Michiel Smid*

July 1, 2001

1 Introduction

Let S be a set of n points in the plane. We want an algorithm for computing a *closest pair* in S, i.e., a pair P, Q of points in S such that

$$d(P,Q) = \min\{d(p,q) : p, q \in S, p \neq q\},$$

where $d(p,q)$ denotes the Euclidean distance between the points p and q. Of course, there may be more than one closest pair. It suffices, however, to find one such pair.

A trivial algorithm works as follows. It considers each pair $p, q \in S$, $p \neq q$, computes its distance, and maintains the smallest distance seen so far. Since there are $\binom{n}{2}$ pairs of distinct points, and one distance can be computed in $O(1)$ time, this algorithm has a running time of $O(n^2)$.

In these notes, we will give a simple closest pair algorithm having an expected running time of $O(n \log n)$. This algorithm is a *randomized incremental algorithm*, which in fact works for higher dimensions as well. It shows the power of *randomization*, in the sense that we get an efficient algorithm that can be understood and implemented very easily.

First some words about *randomization*. An algorithm is randomized if its execution depends on random bits, or if you like, the outcomes of coin tosses. The *running time* for one input is averaged over all possible executions of the algorithm on this input. Hence, the running time is the expected value of a random variable. Then, as usual, the running time is expressed as a function

*Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg. E-mail: michiel@isg.cs.uni-magdeburg.de.

of n, the length of the input. It is defined as the maximal expected running time on any input of length n.

Note that in this way, we get an expected running time that holds for *any* input. There are also probabilistic algorithms that are basically deterministic, but that assume that the input comes from some probability distribution. There, the running time is averaged over *all inputs*. In our model, we average over all possible executions on one single input.

These notes are organized as follows. First, we give a simple algorithm with running time $O(n^2 \log n)$. Then we show that by changing *one line* in this algorithm, we get an *expected* running time of $O(n \log n)$.

2 A slow incremental algorithm

Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in the plane. Our algorithm works *incrementally*, i.e., it successively computes a closest pair in the set $S_i := \{p_1, p_2, \ldots, p_i\}$ for $i = 2, 3, \ldots, n$:

number the points $p_1, p_2, \ldots, p_n$;
$\delta_2 := d(p_1, p_2)$;
$(P, Q) := (p_1, p_2)$;
for $i = 2$ **to** $n - 1$
do (* δ_i is the minimum distance in S_i and
        (P, Q) is a closest pair in S_i *)
    $\delta_{i+1} := \min(\delta_i, d(p_{i+1}, S_i))$;
    update (P, Q) in case $\delta_{i+1} < \delta_i$
endfor;
output (P, Q) and δ_n

How do we implement this algorithm? We can compute the value of δ_{i+1} by first computing

$$d(p_{i+1}, S_i) = \min\{d(p_{i+1}, q) : q \in S_i\},$$

and then taking the minimum of this value and δ_i. Note, however, that if $d(p_{i+1}, S_i)$ is greater than or equal to δ_i, we are not interested in $d(p_{i+1}, S_i)$ at all, because $\delta_{i+1} = \delta_i$ in that case. Hence, we do not always need the value of $d(p_{i+1}, S_i)$.

In order to be able to compute δ_{i+1} efficiently, we extend the algorithm as follows. The set S_i is stored in a δ_i-grid, which is a grid whose cells have sides of length δ_i. Each cell has the form

$$[a\delta_i, (a+1)\delta_i) \times [b\delta_i, (b+1)\delta_i)$$

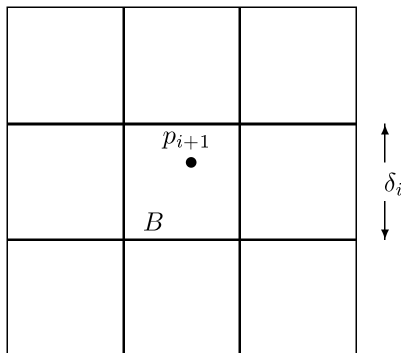

Figure 1: The neighborhood of a grid cell B.

---

for some integers a and b. We call (a, b) the *index* of the cell.

Storing the set S_i in a δ_i-grid means the following. For each point $p = (p_x, p_y)$ of S_i, we compute the index (a_p, b_p) of the grid cell containing this point:

$$a_p = \lfloor p_x/\delta_i \rfloor, b_p = \lfloor p_y/\delta_i \rfloor.$$

This gives a set of $|S_i|$ indices, which we store in a balanced binary search tree. The indices are stored in lexicographical order. With each index, we store a list of all points in S_i that are contained in the cell with this index.

Exercise 1 Convince yourself that given S_i and δ_i, we can compute this data structure in $O(|S_i| \log |S_i|)$ time.

As we will show now, the δ_i-grid has a sparseness property, which will be crucial for the running time of our algorithm.

Lemma 1 *Each cell of the δ_i-grid contains at most four points of S_i.*

Proof. First recall that δ_i is the minimum distance in the set S_i. Assume there is a cell that contains at least five points of S_i. Partition this cell into four subcells, each having sides of length $\delta_i/2$. Then by the pigeon hole principle, there is a subcell containing at least two points of S_i. Let x and y be two such points. Clearly, x and y have distance at most $\sqrt{2} \cdot \delta_i/2 < \delta_i$. This is a contradiction, because all pairs of distinct points in S_i have distance at least δ_i. ■

We need one more notion. The *neighborhood* of a grid cell B is defined as the cell B itself plus the eight cells surrounding B, see Figure 1.

Now we can give the closest pair algorithm. The pseudo code is given in Figure 2, see also Figure 3 for an example.

```
number the points p_1, p_2, ..., p_n;
δ_2 := d(p_1, p_2);
(P, Q) := (p_1, p_2);
store S_2 in a δ_2-grid;
for i = 2 to n − 1
do (* δ_i is the minimum distance in S_i, (P, Q) is a closest pair in S_i,
      and S_i is stored in a δ_i-grid *)
   h := min{d(p_{i+1}, q) : q is in the neighborhood of p_{i+1}'s cell};
   δ_{i+1} := min(δ_i, h);
   if δ_{i+1} < δ_i
   then q := a point in S_i such that d(p_{i+1}, q) = h;
          (P, Q) := (p_{i+1}, q)
   endif;
   (* δ_{i+1} is the minimum distance in S_{i+1} and
      (P, Q) is a closest pair in S_{i+1} *)
   if δ_{i+1} = δ_i
   then add p_{i+1} to the δ_{i+1}-grid
   else discard the δ_i-grid;
        store S_{i+1} in a δ_{i+1}-grid
   endif
endfor;
output (P, Q) and δ_n
```

Figure 2: The incremental closest pair algorithm.

Exercise 2 Prove the correctness of the algorithm. In particular, prove that in the line "$\delta_{i+1} := \min(\delta_i, h)$", the minimum distance of the set S_{i+1} is assigned to the variable δ_{i+1}.

What is the running time of the algorithm? The initialization takes $O(n)$ time, because we have to number the points.

Consider one iteration of the **for**-loop. To compute the value of h, we compute the index (a, b) of the grid cell containing p_{i+1}. Then we search in the balanced binary search tree for the nine indices $(a + \alpha, b + \beta)$, with $\alpha, \beta \in \{-1, 0, +1\}$. For each index found, we compute the distances between p_{i+1} and all points contained in the list stored with this index. By Lemma 1, we compute at most $9 \cdot 4 = 36$ distances.

Since each search in the tree takes $O(\log |S_i|) = O(\log i)$ time, it follows that the value of h can be computed in $O(\log i)$ time. Clearly, the **if-then-endif** part takes only $O(1)$ time.

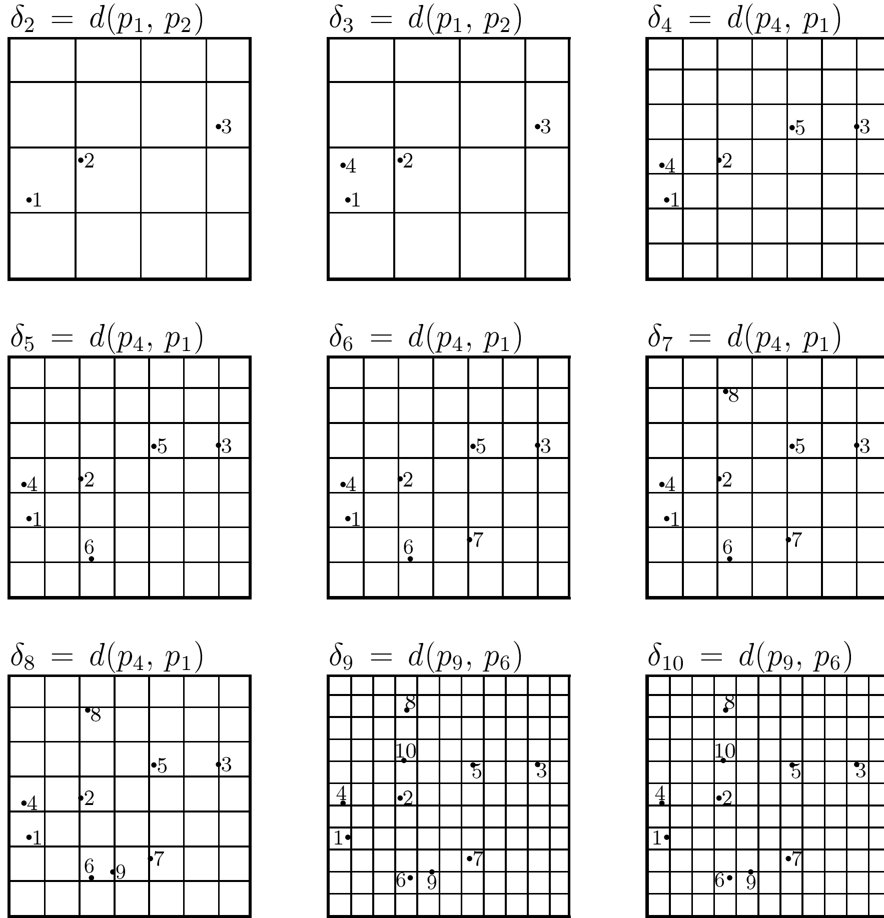

Figure 3: The incremental algorithm running on a set of 10 points. In the beginning, the grid cells have sides of length $d(p_1, p_2)$. Every new minimum distance that is computed during the algorithm causes a refinement of the grid. To avoid clutter, we use i as a shorthand for p_i in the grids.

Consider the **if-then-else-endif** part. If $\delta_{i+1} = \delta_i$, then we search for the index (a, b) of the cell containing p_{i+1}. If this index is stored in the tree, then we add p_{i+1} to the list stored with this index. Otherwise, we insert (a, b) into the tree, and store with it a list containing p_{i+1}. Hence, if $\delta_{i+1} = \delta_i$, we spend $O(\log i)$ time. The **else**-case is the expensive part. If $\delta_{i+1} < \delta_i$, then we store S_{i+1} in a δ_{i+1}-grid, which takes $O(|S_{i+1}| \log |S_{i+1}|) = O((i+1)\log(i+1))$ time.

We have shown that the i-th iteration of the **for**-loop takes

1. $O(\log i)$ time; this is the case if and only if $\delta_{i+1} = \delta_i$,

2. or $O((i+1)\log(i+1))$ time; this is the case if and only if $\delta_{i+1} < \delta_i$.

Therefore, the total worst-case running time of the algorithm is

$$O\left(n + \sum_{i=2}^{n-1} (i+1)\log(i+1)\right) = O(n^2 \log n).$$

Exercise 3 Show that the upper bound of $O(n^2 \log n)$ is tight, i.e., give an input sequence $p_1, p_2, \ldots, p_n$ on which the algorithm actually takes $\Theta(n^2 \log n)$ time.

3 A fast randomized incremental algorithm

As promised, we now improve the running time. Look again at the algorithm. We started with a set S of n points and numbered them $p_1, p_2, \ldots, p_n$. We did not say anything, however, about *how* we number these points. If we number them, e.g., from left to right, then the running time can still be $\Theta(n^2 \log n)$. (Convince yourself that this is true!) Here, *randomization* comes into the picture: In the algorithm in Figure 2, we replace the first line by

Let $p_1, p_2, \ldots, p_n$ be a *random* permutation of the points of S.

We will show that this new randomized algorithm takes $O(n \log n)$ expected time, on *any* input sequence of n points.

Let $T(n)$ be the random variable whose value is equal to the running time of our algorithm on a set S of size n. Then $T(n)$ is the sum of

(1) the total running time for all steps except for the **else**-case, and

(2) the total time for the **else**-case.

The value of (1) is

$$O\left(n + \sum_{i=2}^{n-1} \log i\right) = O(n \log n).$$

What can we say about the expected value of (2)? We know that an execution of the **else**-case in iteration i takes $O((i+1)\log(i+1))$ time. Therefore, in order to get an overall $O(n \log n)$ expected running time, this should not happen too often. The following lemma states that this is indeed the case.

Lemma 2 *The probability that the* **else**-*case is executed during iteration i is at most* $2/(i+1)$.

Proof. The **else**-case is executed if and only if $\delta_{i+1} < \delta_i$. To be more precise, it is executed if and only if

(a) p_{i+1} belongs to some closest pair in S_{i+1}, and

(b) $\delta_{i+1} < \delta_i$.

We compute the probability that (a) and (b) occur.

First assume that S_{i+1} contains only one closest pair, i.e., there is only one pair P, Q of points in S_{i+1} that attains the minimum distance. Then for (a) and (b) to hold, p_{i+1} must be either P or Q.

Otherwise, S_{i+1} contains more than one closest pair. Assume that (a) and (b) hold. Then p_{i+1} must be part of each closest pair of S_{i+1}. (Otherwise, (b) does not hold.) Hence, there is only one possibility for p_{i+1}.

We have shown that there are at most two possibilities for p_{i+1} that force the **else**-case to be executed. But, $p_1, p_2, \ldots, p_n$ is a random permutation of the input sequence. This implies that p_{i+1} is a random element of the set S_{i+1}. Therefore, the probability that the **else**-case is executed is at most $2/|S_{i+1}| = 2/(i+1)$. ■

For each i, $2 \leq i \leq n-1$, let X_i be the random variable whose value is one if the **else**-case is executed during the i-th iteration, and zero otherwise. Then

$$T(n) = O\left(n \log n + \sum_{i=2}^{n-1} X_i \cdot (i+1)\log(i+1)\right),$$

and

$$E(T(n)) = O\left(n \log n + \sum_{i=2}^{n-1} E(X_i) \cdot (i+1)\log(i+1)\right).$$

Note that $E(X_i) = \Pr(X_i = 1)$ and, by Lemma 2, $\Pr(X_i = 1) \leq 2/(i+1)$. Therefore,

$$\begin{aligned} E(T(n)) &= O\left(n \log n + \sum_{i=2}^{n-1} \frac{2}{i+1} \cdot (i+1)\log(i+1)\right) \\ &= O\left(n \log n + \sum_{i=2}^{n-1} \log(i+1)\right) \\ &= O(n \log n). \end{aligned}$$

Theorem 1 *The randomized incremental algorithm computes the closest pair in a set of n points in the plane in $O(n \log n)$ expected time.*

Exercise 4 Extend the algorithm to an arbitrary dimension $D \geq 2$. Show that the expected running time remains $O(n \log n)$, provided D is a constant. (The constant in the Big-Oh bound depends on D.)

The algorithm presented here is an example of a randomized incremental construction. First the n input objects are put in a random order. Then the problem is solved incrementally. There are many more examples of such algorithms: Convex hulls, Voronoi diagrams, intersections of line segments, etc. All these algorithms are relatively easy to analyze and implement, whereas their deterministic counterparts are often much more complicated. Also, the randomized algorithms are often faster in practice than the deterministic ones.