MAX-PLANCK-INSTITUT FÜR INFORMATIK

Simple Randomized Algorithms for Closest Pair Problems

M. Golin R. Raman C. Schwarz M. Smid

MPI–I–92–155 December 1992

Im Stadtwald
W 6600 Saarbrücken
Germany

Simple Randomized Algorithms for Closest Pair Problems

Mordecai Golin* Rajeev Raman† Christian Schwarz†
Michiel Smid†

December 30, 1992

Abstract

We present a conceptually simple, randomized incremental algorithm for finding the closest pair in a set of n points in D-dimensional space, where $D \geq 2$ is a fixed constant. Using dynamic perfect hashing, the algorithm runs in $O(n)$ expected time. In addition to being quick on the average, this algorithm is reliable: we show that it runs in $O(n \log n / \log\log n)$ time with high probability.

1 Introduction

The closest pair problem is to find a closest pair in a given set of points. The problem has a long history in computational geometry and has been extensively studied. It is well known, for example, that finding the closest pair in a set of n points requires $\Omega(n \log n)$ time in the algebraic decision tree model of computation [1] and that there are optimal algorithms which match this lower bound. It is also well known that if the model of computation is changed then the lower bound no longer holds. This was first shown by Rabin [9] who described an algorithm that combines the use of the floor function with randomization to achieve an $O(n)$ expected running time. The expectation is taken over choices made by the algorithm and not over possible inputs. Recently, Khuller and Matias [6] have described a radically different algorithm that also uses the floor function and randomization to achieve an $O(n)$ expected running time.

In this paper, we present yet another algorithm that combines the use of the floor function and randomization to solve the problem in $O(n)$ expected time. Our algorithm is conceptually simpler than the ones in [9] and [6]. It also differs from them in that it is a *randomized incremental algorithm*. The other two algorithms are inherently static. The algorithm that we present here assumes that the input points are fed to it in a sequence which is a random permutation $p_1, p_2, \ldots, p_n$ of the n points. The i-th stage of the

*INRIA-Rocquencourt, 78153 Le Chesnay Cedex, France, and Hongkong UST. This author was supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II) and by NSF grant CCR-8918152. Part of this work was done while this author was visiting the Max-Planck-Institut für Informatik.

†Max-Planck-Institut für Informatik, W-6600 Saarbrücken, Germany. These authors were supported by the ESPRIT Basic Research Actions Program, under contract No. 7141 (project ALCOM II).

algorithm will find the closest pair of the set $S_i := \{p_1, p_2, \ldots, p_i\}$ in $O(1)$ expected time leading to an overall $O(n)$ expected running time. To prove this running time, we use the technique of backwards analysis, due to Seidel. See [12, 13].

In Section 2, we present the algorithm and its analysis. We give an $O(n \log n)$ expected time tree based algorithm and show how to use dynamic perfect hashing [3] to improve its expected running time to $O(n)$.

In Section 3, we consider the problem of returning not only the closest pair, but all the k closest pairs, where k is an integer between 1 and $\binom{n}{2}$. For this problem, there are deterministic algorithms with running time $O(n \log n + k)$, which is optimal in the algebraic decision tree model of computation. (See [7, 10].) Combined with randomization and the floor function, we get a simple algorithm with expected running time $O(k\, n)$, which is better than the algorithms in [7, 10] if $k = o(\log n)$.

In Section 4, we show that our closest pair algorithms are reliable: For example, the linear expected time algorithm actually runs in $O(n \log n / \log \log n)$ time with high probability.

All algorithms of this paper assume that the floor function can be computed at unit cost. In Section 5, however, we give a variant of the closest pair algorithm presented in Section 2 that has $O(n \log n)$ expected running time, even with high probability, without using this non-algebraic function. This algorithm fits in the algebraic decision tree model of computation, extended with the power of randomization.

2 The closest pair algorithm

To keep our exposition simple, we give the algorithm for the two-dimensional case and the euclidean metric. The extension to arbitrary, but fixed, dimension $D \geq 2$ and arbitrary L_t-metric, $1 \leq t \leq \infty$, is straightforward. Let $d(p, q)$ denote the distance between the points $p = (p^{(1)}, p^{(2)})$ and $q = (q^{(1)}, q^{(2)})$, i.e.,

$$d(p, q) = \sqrt{(p^{(1)} - q^{(1)})^2 + (p^{(2)} - q^{(2)})^2}.$$

Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of points. The *closest pair distance* in S is

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

The *closest pair problem* is to find a pair of points $p, q \in S$ such that $d(p, q) = \delta(S)$.

Our algorithm will be based upon the following simple observation. Let $S_i := \{p_1, p_2, \ldots, p_i\}$ be the set containing the first i points of S. Then $\delta(S_{i+1}) < \delta(S_i)$ if and only if there is some point $p \in S_i$ such that $d(p, p_{i+1}) < \delta(S_i)$.

Suppose a square grid with mesh size $\delta(S_i)$ is laid over the plane[1] and each point of S_i is stored in the grid box in which it appears. Let b be the grid box in which the new point p_{i+1} is located. Then every point in S_i that is within distance $\delta(S_i)$ of p_{i+1} must be located in one of the 9 grid boxes that are adjacent to b. (We consider the box b as being adjacent to itself.) We call these 9 boxes the *neighbors* of b.

We note that each grid box can only contain at most four points from S_i. This is because if a grid box contained more than four points then some pair of them would be less than $\delta(S_i)$ apart, contradicting the definition of $\delta(S_i)$.

[1] To exactly specify the grid we will always assume that $(0, 0)$ is one of its lattice points.

The above observations lead to the following generic algorithm for finding the closest pair in S. The points will be fed to the algorithm in random order $p_1, p_2, \ldots, p_n$, i.e., each of the $n!$ possible orders is equally likely. The algorithm starts by calculating $\delta(S_2) = d(p_1, p_2)$ and inserting the points of S_2 into the grid with mesh size $\delta(S_2)$. It then proceeds incrementally, always keeping set S_i stored in a grid with mesh size $\delta(S_i)$. When fed point p_{i+1} it finds the at most 36 points in the 9 grid boxes neighboring the box in which p_{i+1} is located and computes d_{i+1}, the minimum distance between p_{i+1} and these at most 36 points. If there are no points in these boxes then $d_{i+1} = \infty$. From the discussion above we know that $\delta(S_{i+1}) = \min(d_{i+1}, \delta(S_i))$.

If $d_{i+1} \geq \delta(S_i)$ then $\delta(S_{i+1}) = \delta(S_i)$ and the algorithm inserts p_{i+1} into the current grid. Otherwise, $\delta(S_{i+1}) = d_{i+1} < \delta(S_i)$ and the algorithm discards the old grid, creates a new one with mesh size $\delta(S_{i+1})$, and inserts the points of S_{i+1} into this grid.

The algorithm thus calculates $\delta(S_i)$ for $i = 2, 3, \ldots, n$, in this order. Then it outputs the value $\delta(S_n) = \delta(S)$. An example of our algorithm is shown in Figure 1.

Remark: Instead of one closest pair, we can also incrementally maintain a list of all point pairs in S_i with distance $\delta(S_i)$, for $2 \leq i \leq n$, without additional cost. Consider the i-th stage of the algorithm, when we add p_{i+1} to S_i. Since the algorithm checks all points within distance $\delta(S_i)$ of p_{i+1} anyway, we find all points q in S_i such that $d(q, p_{i+1}) = \delta(S_i)$. The pairs (q, p_{i+1}) with $\delta(q, p_{i+1}) = \delta(S_{i+1})$ — there can be only $O(1)$ of them — are exactly the new closest pairs after the i-th stage of the algorithm. If $\delta(S_{i+1}) = \delta(S_i)$, we add these pairs to the current closest pair list. If $\delta(S_{i+1}) < \delta(S_i)$, we discard the old list and make a new list that contains just these pairs.

To actually implement the above algorithm we will need the following: let P be a point set, d a positive real number, p a point, $\mathcal{G}$ a grid, and b the name of a box in a grid. We define the following operations.

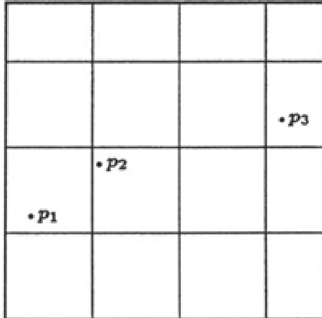
- *Build*(P, d) : Return a grid $\mathcal{G}$ with mesh size d that contains the points in P.
- *Insert*$(\mathcal{G}, p)$: Insert point p into grid $\mathcal{G}$.
- *Report*$(\mathcal{G}, b)$: Return all points in grid box b.

Pseudocode for the closest pair algorithm using these operations is presented in Figure 2.

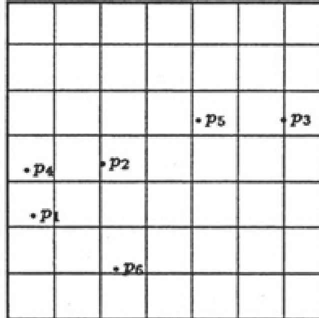
How do we actually implement these grid operations? One easy way is to use balanced binary search trees: Consider a grid $\mathcal{G}$ with mesh size d, let $p = (p^{(1)}, p^{(2)})$ be a point in the plane, and denote the box containing p in $\mathcal{G}$ by b_p. The integer pair $(\lfloor p^{(1)}/d \rfloor, \lfloor p^{(2)}/d \rfloor)$ is called the *index* of b_p. The point set is stored as follows: We determine the indices of the non-empty boxes and store them in lexicographical order in a balanced binary search tree. Moreover, with each box in this tree, we store a list of all points that are contained in this box. To insert a point into the grid, we use the floor function to compute the index of the grid box that contains this point. Then we search in the tree for this box. If it is stored in the tree, then we insert the new point into the list that is stored with the box. Otherwise, we create a new node to hold the new grid box, together with a list containing the new point.

Using this implementation, it takes $O(n \log n)$ time to run *Build*(P, d) for $|P| = n$. When $\mathcal{G}$ stores n points, then *Insert*$(\mathcal{G}, p)$ will cost $O(\log n)$ time and *Report*$(\mathcal{G}, b)$ will cost $O(\log n + |b \cap P|)$ time.

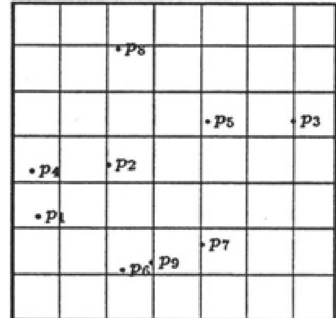
$\delta(S_2) = d(p_1, p_2)$



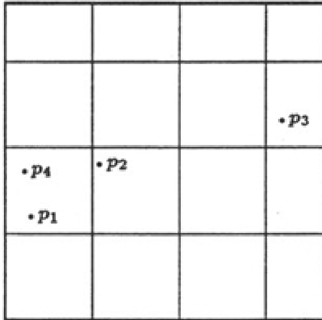
$\delta(S_5) = d(p_4, p_1)$



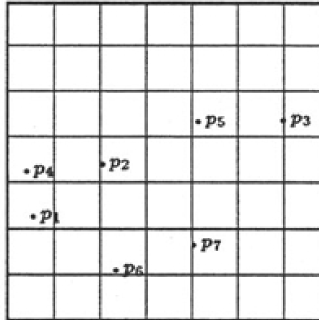
$\delta(S_8) = d(p_4, p_1)$



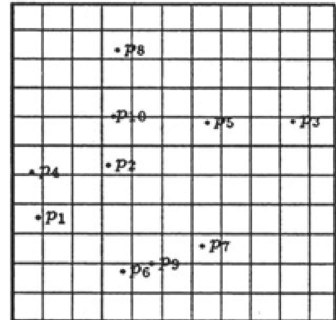
$\delta(S_3) = d(p_1, p_2)$



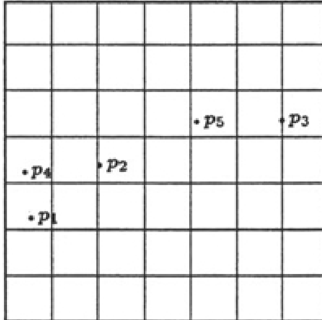
$\delta(S_6) = d(p_4, p_1)$



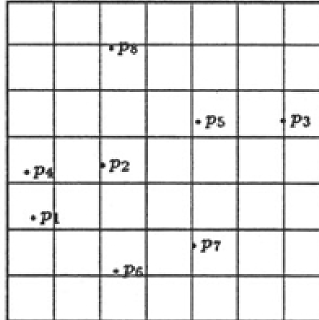
$\delta(S_9) = d(p_9, p_6)$



$\delta(S_4) = d(p_4, p_1)$



$\delta(S_7) = d(p_4, p_1)$



$\delta(S_{10}) = d(p_9, p_6)$

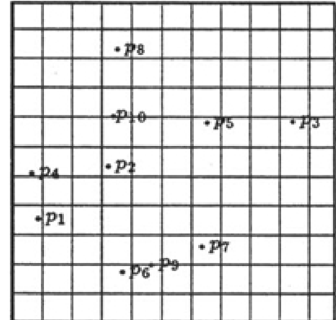


Figure 1: The incremental algorithm running on a set of 10 points. In the beginning the grid has mesh size $d(p_1, p_2)$. Every new minimal distance that is computed during the algorithm causes a refinement of the grid.

Algorithm $CP(p_1, p_2, \ldots, p_n)$

```
(1) δ := d(p_1, p_2); G := Build(S_2, δ);
(2) for i := 2 to n − 1 do
(3)     begin
(4)     V := {Report(G, b) : b is a neighbor of the box containing p_{i+1}};
(5)     d := min_{q∈V} d(p_{i+1}, q);
(6)     if d ≥ δ then Insert(G, p_{i+1})
(7)             else δ := d; G := Build(S_{i+1}, δ);
(8)     end;
(9) return(δ).
```

Figure 2: Pseudocode for the closest pair algorithm.

Assume the points are available in two lists X and Y, sorted by x- and y-coordinates, respectively. Moreover, assume that each point in Y contains a pointer to its occurrence in X. Then the running time of $Build(P, d)$ can be improved to $O(n)$: Walk along the list X and compute the value $\lfloor p^{(1)}/d \rfloor$ for each point p. Initialize an empty bucket $B(\lfloor p^{(1)}/d \rfloor)$ for all distinct values $\lfloor p^{(1)}/d \rfloor$. Moreover, give each point in X a pointer to its bucket. Then, go through the list Y. For each point in this list, follow the pointer to its occurrence in X and, from there, follow the pointer to its bucket and store the point at the end of this bucket. Finally, concatenate all buckets into one list. This list contains the indices of all non-empty boxes in the grid $\mathcal{G}$, sorted in lexicographical order.

Another way to implement the grid operations is by using dynamic perfect hashing [3] to store the currently non-empty grid boxes. Then, we can implement *Build* in $O(n)$ expected time, *Insert* in $O(1)$ expected time and *Report* in $O(1 + |b \cap P|)$ deterministic time.

We should point out that dynamic perfect hashing does not permit the insertion of totally arbitrary items into a lookup table. It requires that the universe containing the items be known in advance, and that we have a prime exceeding the size of the universe. In terms of grids, the first requirement translates into having a bound on the indices of possible non-empty grid boxes. As mentioned above, the index of the box containing $p = (p^{(1)}, p^{(2)})$ in a grid with mesh size d is the integer pair $(\lfloor p^{(1)}/d \rfloor, \lfloor p^{(2)}/d \rfloor)$. Therefore, the set of integers $\{\lfloor p_i^{(j)}/d \rfloor, 1 \le i \le n, j = 1, 2\}$ should come from a bounded universe. This is the case in our application because we know all points in advance. So, if we have to build a grid for the current set with a given d we have a bound on the indices of all non-empty grid boxes *before* we start gridding. Therefore, our method works for all input sets where the size of the universe that contains the indices of the grid boxes is moderate enough such that the time spent on finding a suitable prime does not affect the overall running time of the building algorithm.

We now analyze the cost of both the tree and hashing based implementation of the algorithm. Let i be fixed. Line 4 of the algorithm will call *Report*() 9 times to find at most 36 points. Therefore lines 4-5 will use $O(\log i)$ deterministic time in the tree based

implementation and $O(1)$ deterministic time in the hashing based one.

Line 6 will be called at most once and uses $O(\log i)$ deterministic time for the tree based implementation, and $O(1)$ expected time if we use hashing.

Line 7 will use $O(i \log i)$ deterministic time if we use trees and $O(i)$ expected time if we use hashing. We can improve the bound for trees to $O(i)$ also if we maintain the points of S_i sorted by both their coordinates. Then, as discussed before, the procedure *Build* runs in linear time. Note that line 7 is called if and only if $\delta(S_{i+1}) < \delta(S_i)$. We will prove in Lemma 1 that this happens with probability at most $2/(i+1)$. Therefore, the expected cost of line 7 is $O(1)$ for both the tree based and the hashing based implementation. (For the hashing based implementation, the expected cost of line 7 is composed of two random variables: One variable indicating if we build a new grid, and the other variable gives the time needed for this. This second variable depends on coin tosses that are made to build a hash table. These coin tosses have nothing to do with the first random variable. Therefore, the two random variables are independent. Hence, the expected cost of line 7 is $O(i) \cdot 2/(i+1) = O(1)$.)

Combining the three paragraphs above proves that the i-th stage of our closest pair algorithm runs in $O(1)$ expected time if we use dynamic perfect hashing. If the points are maintained in sorted order w.r.t. each coordinate, the i-th stage of the tree based algorithm takes $O(\log i)$ deterministic time plus $O(1)$ expected time. The latter separation of deterministic time and expected time will be crucial for the "high probability" running time of the algorithm, which will be discussed in Section 4.

So, to find $\delta(S) = \delta(S_n)$, the algorithm uses $O(n \log n)$ expected time for the tree based implementation, and $O(n)$ expected time if we use dynamic perfect hashing.

As mentioned at the beginning of this section, our algorithm works for points in any dimension and for any L_t-metric, $1 \le t \le \infty$. Suppose S is a collection of D-dimensional points, where $D \ge 2$. We modify the algorithm by extending the definition of a grid to be D-dimensional and define the neighbors of a grid box to be the 3^D grid boxes that adjoin it. The algorithm and analysis proceed as before. Note that a box in a grid with mesh size $\delta(S_i)$—which now is the minimal L_t-distance in S_i—contains at most $(D+1)^D$ points of S_i. (See [11].)

We mention here that our algorithm, whatever implementation we take, is much simpler than the known $O(n \log n)$ deterministic algorithms for finding the closest pair in dimensions higher than two. (See [2, 7, 10, 11] for some of such algorithms.)

Lemma 1 *Let $p_1, p_2, \ldots, p_n$ be a random permutation of the points of S. Let $S_i := \{p_1, p_2, \ldots, p_i\}$. Then* $\Pr[\delta(S_{i+1}) < \delta(S_i)] \le 2/(i+1)$.

Proof: We use Seidel's backwards analysis technique. (See [12, 13].) Consider S_i, p_{i+1} and $S_{i+1} = S_i \cup \{p_{i+1}\}$. Let

$$A := \{p \in S_{i+1} \,:\, \exists q \in S_{i+1} \text{ such that } d(p,q) = \delta(S_{i+1})\},$$

i.e., A is the set of points that are part of some closest pair in S_{i+1}. If $|A| = 2$ then there is exactly one closest pair in S_{i+1} and

$$\delta(S_{i+1}) < \delta(S_i) \iff p_{i+1} \in A.$$

If $|A| > 2$ there are two possibilities. The first is that there is a unique $p \in A$ that is a member of every closest pair in S_{i+1}. In this case

$$\delta(S_{i+1}) < \delta(S_i) \Longleftrightarrow p_{i+1} = p.$$

The other possibility is that there is no such unique p. In that case, S_i must contain some pair of points from A and, therefore, $\delta(S_{i+1}) = \delta(S_i)$.

We have just shown that, regardless of the composition of S_{i+1}, there are at most 2 possible choices of p_{i+1} which will permit $\delta(S_{i+1}) < \delta(S_i)$. Since $p_1, p_2, \ldots, p_n$ is a random permutation, the point p_{i+1} is a random point from S_{i+1}. Therefore, the probability that $\delta(S_{i+1})$ is smaller than $\delta(S_i)$ is at most $2/(i+1)$. ∎

We summarize our result:

Theorem 1 *Let S be a set of n points in D-space, and let $1 \leq t \leq \infty$.*

1. *The implementation of the algorithm that uses a binary tree finds a closest pair in S, in $O(n \log n)$ expected time. This implementation uses the floor function and randomization, and it works for any set S.*
2. *The implementation of the algorithm that uses dynamic perfect hashing finds a closest pair in S, in $O(n)$ expected time. This implementation uses the floor function and randomization, and it works for any set S for which the values $\lfloor p^{(j)}/\delta(S) \rfloor$, $p = (p^{(1)}, p^{(2)}, \ldots, p^{(n)}) \in S$ and $1 \leq j \leq D$, come from a finite universe.*

3 The k closest pairs problem

In this section we quickly sketch how to modify our algorithm to provide a simple solution to the k closest pairs problem. Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of n points in the plane. Enumerate all $\binom{n}{2}$ distances between pairs of points and sort them as $e_1 \leq e_2 \leq \ldots \leq e_{\binom{n}{2}}$. Set $\delta^k(S) := e_k$ to be the *k-th closest pair distance* in the set S. *The k closest pairs problem* is to find k pairs of points that are at most $\delta^k(S)$ apart.

Let $p_1, p_2, \ldots, p_n$ be given to us in a random order. Let $S_i = \{p_1, p_2, \ldots, p_i\}$. Our algorithm will incrementally calculate $\delta^k(S_i)$. It differs from the closest pair algorithm only in that it uses a mesh size of $\delta^k(S_i)$ instead of $\delta(S_i)$ for the grid that stores S_i.

Assume the algorithm has already seen the first i points. Moreover, assume that the algorithm

1. has stored S_i in a grid with mesh size $\delta^k(S_i)$, and
2. has a binary search tree (sorted in increasing order by distance) containing k pairs of points from S_i that are at most $\delta^k(S_i)$ apart.

The algorithm now performs the i-th stage, i.e., it gets p_{i+1} and wants to update the information it is storing.

It does this by finding all points within distance $\delta^k(S_i)$ of p_{i+1}. Note that all these points must be in one of the 9 grid boxes neighboring p_{i+1}. Also note that each grid box can contain at most $8\sqrt{k}$ points from S_i: If a box contained more than $8\sqrt{k}$ points,

then more than k points would be less than distance $\delta^k(S_i)$ from each other which is impossible. We can therefore use 9 *Report*() operations as described in the previous section to find all of the at most $72\sqrt{k}$ points within these boxes. The algorithm then calculates all distances between these points and p_{i+1} and inserts them into the tree that stores the k smallest distances. For each inserted distance, we delete the maximal element stored in the tree. In this way, the tree still contains k elements, which form the k closest pairs in S_{i+1}. Moreover, $\delta^k(S_{i+1})$ is known at this moment. All this takes $O(\sqrt{k}\log k)$ time.

If $\delta^k(S_{i+1}) = \delta^k(S_i)$ then the algorithm inserts p_{i+1} into the current grid. If $\delta^k(S_{i+1}) < \delta^k(S_i)$ then the algorithm discards the current grid and inserts all points of S_{i+1} into a new grid with mesh size $\delta^k(S_{i+1})$.

An analysis similar to the one performed in the previous section shows that—if the points are fed to the algorithm in a random order—a new grid is built in the i-th stage with probability at most $2k/(i+1)$. Therefore, the expected cost of inserting the $(i+1)$-st point will be $O(k + \log i)$ if trees are used and $O(k)$ if dynamic perfect hashing is used. Thus, the algorithm finds $\delta^k(S) = \delta^k(S_n)$ in $O(n(k + \log n))$ expected time in tree based implementations and $O(nk)$ expected time in hashing based ones.

Theorem 2 *The tree-based implementation of the k closest pairs algorithm runs in $O(n(k + \log n))$ expected time, whereas the hashing-based implementation runs in $O(nk)$ expected time.*

4 High probability bounds

In this section we will prove that the closest pair algorithm runs quickly with high probability. To achieve this result, we apply a method due to Clarkson, Mehlhorn and Seidel [4] for obtaining tail estimates on the space complexity of some randomized incremental constructions, and a dynamic perfect hashing scheme due to Dietzfelbinger and Meyer auf der Heide [3].

In each iteration of the closest pair algorithm of Figure 2, some (relatively cheap) work is done no matter which point is added or which points have been added before, such as inserting the new point into the data structure or computing the new closest pair. More interesting for the probabilistic analysis is the expensive rebuilding operation that has to be performed — depending on the point that is added and the points that have been added before — with low probability. Therefore, we study a random variable that describes this rebuilding cost.

Definition 1 For any set T of points and any point $p \in T$, define

$$cost(p, T) = \begin{cases} |T| & \text{if } \delta(T) < \delta(T \setminus \{p\}) \\ 0 & \text{otherwise.} \end{cases}$$

That is, if we already have computed $\delta(T \setminus \{p\})$, then $cost(p, T)$ expresses the rebuilding cost of the closest pair algorithm when computing $\delta(T)$.

Let S be a set of n points. We define a random variable Y_S as follows: Let $p_1, p_2, \ldots, p_n$ be a random permutation of the set S and let $S_i = \{p_1, p_2, \ldots, p_i\}$ for $1 \leq i \leq n$. Then the random variable Y_S has value $Y_S = \sum_{i=3}^{n} cost(p_i, S_i)$.

Lemma 2 *For all $c \geq 1$,*

$$\Pr[Y_S \geq cn] \leq \frac{1}{e^2}\left(\frac{e^2}{c}\right)^c.$$

Our proof of this tail estimate follows the general line of the tail estimate proof in [4]. We will obtain a bound on the probability generating function of Y_S and use this to obtain a bound on the probability that Y_S exceeds the value cn.

Definition 2 Let Z be a non-negative random variable that takes only integer values. The *probability generating function (pgf)* of Z is defined by

$$G_Z(x) = \sum_{j \geq 0} \Pr[Z = j] \cdot x^j.$$

Claim 1 *For any $h \geq 0$ and $a \geq 1$*

$$\Pr[Z \geq h] \leq G_Z(a)/a^h.$$

Proof: $G_Z(a) = \sum_{j \geq 0} \Pr[Z = j] \cdot a^j \geq \sum_{j \geq h} \Pr[Z = j] \cdot a^j \geq a^h \sum_{j \geq h} \Pr[Z = j]$. ∎

By this fact, we can use bounds on the pgf of Z to obtain a tail estimate for Z.

Now let us look at the pgf $G_{Y_S}(x)$ of our random variable Y_S. We will use $G_S(x)$ as a short form for $G_{Y_S}(x)$.

Claim 2 *For all $x \geq 1$*

$$G_S(x) \leq p_n(x) := \prod_{1 \leq i \leq n} \left(1 + \frac{2}{i}(x^i - 1)\right).$$

Proof: The proof is by induction on n, the size of S. For $n = 1$ and 2, the claim holds, because then $G_S(x) = 1$ and the product on the right-hand side is at least equal to one.

Let $n \geq 3$ and assume the claim holds for $n - 1$. Since $p_1, p_2, \ldots, p_n$ is a random permutation of S, p_n is random element of S, and so

$$G_S(x) = \frac{1}{n} \sum_{p \in S} x^{cost(p,S)} G_{S \setminus \{p\}}(x).$$

Applying the induction hypothesis yields

$$G_S(x) \leq \frac{p_{n-1}(x)}{n} \sum_{p \in S} x^{cost(p,S)}.$$

From Lemma 1 we know that there are at most two points p in S such that $cost(p, S) = n$. For the other points p, $cost(p, S) = 0$. Therefore,

$$G_S(x) \leq \frac{p_{n-1}(x)}{n}(2x^n + n - 2) = p_{n-1}(x)\left(1 + \frac{2}{n}(x^n - 1)\right) = p_n(x).$$

∎

Proof of Lemma 2: We apply the above claims:

$$\begin{aligned}
\Pr[Y_S \geq cn] &\leq G_S(a)/a^{cn} && \text{for any } a \in \mathbb{R}_{\geq 1} \text{ by Claim 1} \\
&\leq \left(\prod_{1\leq i\leq n}\left(1+\tfrac{2}{i}(a^i-1)\right)\right)/a^{cn} && \text{by Claim 2} \\
&\leq \exp\left(\sum_{1\leq i\leq n}\tfrac{2}{i}(a^i-1)\right)/a^{cn} && \text{since } 1+x\leq e^x \\
&\leq \exp(2(a^n-1))/a^{cn} && \text{since } \tfrac{2}{i}(a^i-1)\leq \tfrac{2}{n}(a^n-1) \\
& && \text{for each } i\leq n \text{ and each } a\geq 1 \\
&= \tfrac{1}{e^2}\left(\tfrac{e^2}{c}\right)^c && \text{with } a=c^{1/n}.
\end{aligned}$$

∎

We can now analyze the closest pair algorithm, first turning our attention to the tree based implementation. The i-th stage of the algorithm requires $O(\log i)$ time for searching out points in neighboring boxes, inserting p_{i+1} into the lists that maintain the points sorted by all their coordinates, and (possibly) inserting p_{i+1} into the grid. If $\delta(S_{i+1}) < \delta(S_i)$ then it will regrid the points in $O(i)$ time. Thus the full work done by the i-th stage of the algorithm is described by $O\left(\log i + cost(p_{i+1}, S_{i+1})\right)$ and the total work performed by the algorithm is $O\left(n \log n + Y_S\right)$.

Let s be a positive integer. We apply Lemma 2 with $c = 2 \cdot s \cdot \ln n / \ln\ln n$. Then, for n sufficiently large, we have $2c - c\ln c \leq -s \ln n$ and therefore

$$\Pr[Y_S \geq 2sn\ln n/\ln\ln n] \leq \frac{1}{e^2}\left(\frac{e^2}{c}\right)^c = \frac{1}{e^2}e^{2c-c\ln c} \leq \frac{1}{e^2}e^{-s\ln n} = O(n^{-s}).$$

This shows that $Y_S = O(n\log n/\log\log n)$ with probability $1-O(n^{-s})$ for every s. That is, the tree based implementation runs in $O(n\log n)$ time with probability $1-O(n^{-s})$ for any positive integer s.

Now let us analyze the hashing based implementation. We need some facts from [3] about their dynamic perfect hashing scheme. Their scheme can build[2] a hash table for n items in $O(n)$ time with probability $1-O(n^{-s})$ for every s. Moreover, a new element can be inserted into a hash table storing n items in time $O(1)$, also with probability $1-O(n^{-s})$.

We assume a slight variant of the closest-pair algorithm. This variant does not start with $i=2$ but instead uses a brute force method to find $\delta(S_{\sqrt{n}})$ in $O(n)$ time and inserts $S_{\sqrt{n}}$ into the grid with mesh size $\delta(S_{\sqrt{n}})$. Only then, with $i=\sqrt{n}, \sqrt{n}+1, \ldots, n$ will it start running the incremental algorithm.

Making this change ensures that the hash table always stores at least $\sqrt{n}$ items. Therefore, with probability $1-O(n^{-s/2})$, an insert into the table will take only $O(1)$ time. Moreover, a rebuild on i items will take $O(i)$ time with probability $1-O(n^{-s/2})$. We can therefore assume that, with probability $1-O(n^{1-s/2})$, over the entire algorithm, every insert takes $O(1)$ time and every rebuild on i elements takes $O(i)$ time. That is, with probability $1-O(n^{1-s/2})$, the total work performed by the algorithm is $O(n+\sum_i cost(p_{i+1}, S_{i+1})) = O(n+Y_S)$. We saw already that $Y_S = O(n\log n/\log\log n)$ with probability $1-O(n^{-s})$.

---

[2]The algorithm presented in [3] does not explicitly show how to build a hashtable with this probability. It can be modified to do so, though, without too much difficulty.

This shows that for each $s \geq 1$, the algorithm runs in $O(n \log n / \log\log n)$ time with probability $1 - O(n^{1-s})$.

We summarize our results:

Theorem 3 *The implementation of the closest pair algorithm that uses a binary tree runs in $O(n \log n)$ time with probability $1 - O(n^{-s})$ for every s. The hashing-based implementation of the closest pair algorithm runs in $O(n \log n / \log\log n)$ time with probability $1 - O(n^{-s})$ for every s.*

5 An algebraic decision tree implementation

In the previous sections, we stored the non-empty grid boxes using binary trees or perfect hashing. Both implementations, however, use the non-algebraic floor function: We need this function for computing the grid box that contains a given point. It is well known that the floor function is very powerful: For the maximum-gap problem, there is an $\Omega(n \log n)$ lower bound for the algebraic decision tree model. Adding the floor function, however, leads to an $O(n)$ time complexity. (See [8].)

In this section, we sketch how to obtain an $O(n \log n)$ expected time algorithm for the closest pair problem that does not use the floor function and that runs on any set S. The techniques appear already in [5, 7]. For simplicity, we restrict ourselves to the planar case. The main idea is to use a slightly degraded grid:

Definition 3 *Let S be a set of n points in the plane and let d be a positive real number. Suppose we have the following:*

1. *Let $s_1, s_2, \ldots$ be a sequence of pairwise disjoint vertical slabs of width between d and $2d$, such that each slab contains at least one point of S, and each point of S is contained in one of these slabs. The s_i's induce vertical slabs that do not contain points of S. These empty slabs have width either zero or at least d.*

2. *For $i \geq 1$, let V_i be the set of points in S that are contained in slab s_i. Let $b_{i1}, b_{i2}, \ldots$, be a sequence of pairwise disjoint boxes having width equal to the width of slab s_i, and having height between d and $2d$, such that each box contains at least one point of V_i, and each point of V_i is contained in one of these boxes. The b_{ij}'s induce boxes in s_i that do not contain points of S. These boxes have height either zero or at least d.*

The set of non-empty boxes—for all i together—is called a d-covering *of S.*

Figure 3 shows an example. A d-covering can be computed in $O(n \log n)$ time by a sweep algorithm. In fact, if we maintain the points of S in two binary search trees, one tree containing the points in sorted x-order, the other in sorted y-order, and if each point in one tree contains a pointer to its occurrence in the other tree, then a d-covering can be computed in $O(n)$ time, in a way similar to the linear time algorithm for the grid sketched in Section 2. (See [7].) During the construction, we do not use the floor function.

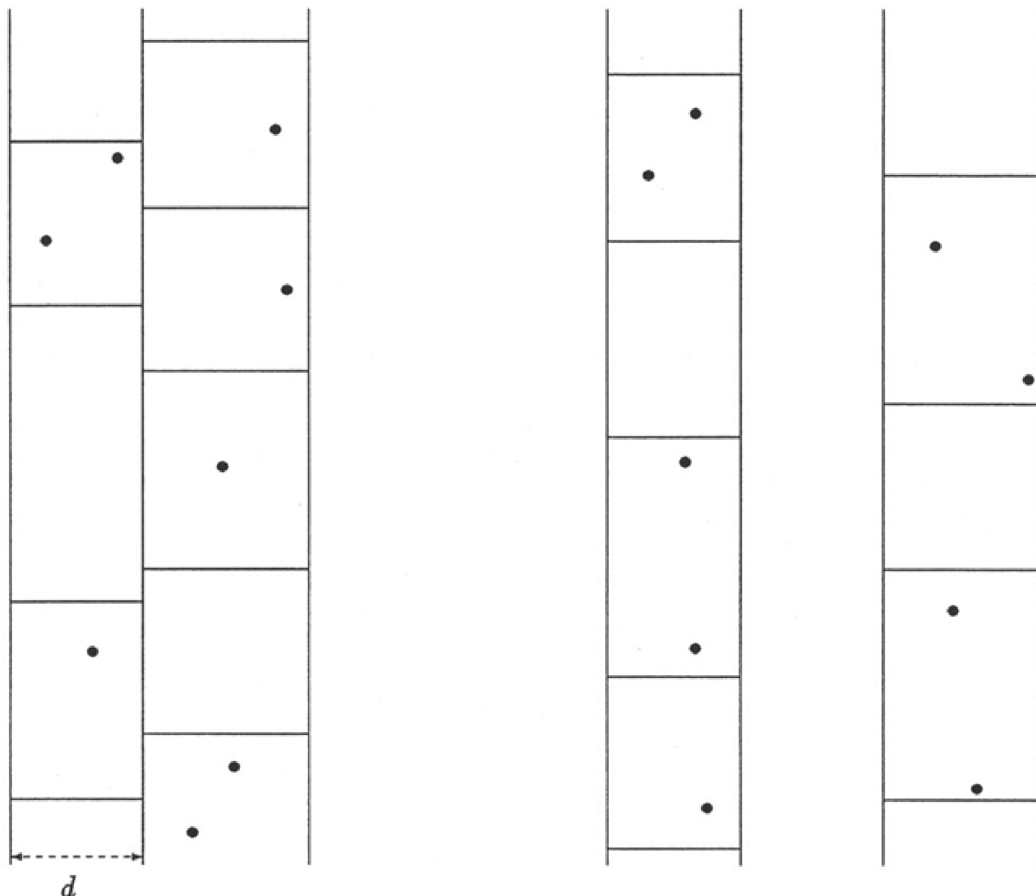


Figure 3: A d-covering for a planar point set. The rectangles that contain a point have sidelengths between d and $2d$.

It follows from Definition 3, that we can store a d-covering using the slab method. Hence, we can locate the box containing a point p in $O(\log n)$ time, without using the floor function.

Suppose we want to insert a new point p. Then, we first check if there is already a non-empty box containing p. If so, we insert p to the list stored with this box. Otherwise, we have to insert a new box. We distinguish two cases.

First assume that the slab containing p is empty. The search for p ended between two non-empty slabs, say s and t. If the slab between s and t has width less than $2d$, we make one new (non-empty) slab out it. Moreover, we let the new box start at the y-coordinate of p, and let its height be d. We insert the new slab and the new box into the data structure. Otherwise, the slab between s and t has width at least $2d$. We make a new slab, say u, taking care that the slab between s and u has width either zero, or at least d. Similarly, the slab between u and t has width either zero or at least d. As before, we make in slab u a new box—of height d—containing p, and we insert u and the new box into the data structure.

The case that the slab containing p is already non-empty, but there is no non-empty box in this slab containing it, can be handled similarly.

The entire insert algorithm takes $O(\log n)$ time. The delete algorithm, which also takes $O(\log n)$ time, is similar.

The data structure looks as follows. We store the points of S_i in a $\delta(S_i)$-covering. Moreover, we maintain the two binary search trees that allow us to construct the covering in linear time. Note that, as it was the case for the grid with mesh size $\delta(S_i)$, each box of the $\delta(S_i)$-covering contains only $O(1)$ points of S_i—a fact that is crucial to the running time of the algorithm, see Section 2.

We have to redefine the notion of neighborhood. Let b be a box of a d-covering R. Let $(b^{(1)}, b^{(2)})$ be the coordinates of the lower left corner of b. Then, the *neighbors* of b are all boxes in R that overlap the square

$$s := [b^{(1)} - d : b^{(2)} + 3d] \times [b^{(1)} - d : b^{(2)} + 3d].$$

The boxes that overlap s can be found by locating the 25 points

$$(b^{(1)} + \alpha d, b^{(2)} + \beta d), \quad \text{for } \alpha, \beta \in \{-1, 0, 1, 2, 3\}$$

in the structure that stores the covering R.

Theorem 4 *Let S be a set of n points in D-space, and let $1 \leq t \leq \infty$. There exists an algorithm that incrementally computes a closest pair in S in $O(n \log n)$ expected time. This algorithm fits in the algebraic decision tree model of computation extended with the power of randomization. Moreover, this algorithm runs in $O(n \log n)$ time with probability $O(n^{-s})$ for every s.*

Proof: Since we maintain the points sorted w.r.t. all their coordinates, we can build a $\delta(S_i)$-covering in $O(i)$ time. That is, for the i-th stage, we need $O(\log i)$ deterministic time for updating the trees and for the point location in the covering, plus $O(1)$ expected time because of the rebuilding. It follows that the expected as well as the "high probability" running time is $O(n \log n)$, see Sections 2 and 4. ∎

Remark: Note that the above algorithm is of the Las Vegas type, i.e. it always computes a correct output — the running time of the algorithm is a random variable. The $\Omega(n \log n)$ for the element uniqueness problem in the algebraic decision tree model (from which the lower bound for the closest pair problem follows by reduction) not only holds for the worst case, but also for the expected running time of an input drawn from a uniform distribution, see [1]. A Las Vegas algorithm is a deterministic algorithm for any fixed sequence of coin flips. So, for any distribution of inputs, the average running time of a Las Vegas algorithm — where the average is taken over the inputs of the distribution and over the coin flips — cannot be smaller than the best average running time that is achievable by a deterministic algorithm for this distribution. Since the average case complexity of a Las Vegas algorithm for some input distribution cannot be larger than the worst case (w.r.t. inputs) running time of this algorithm, $\Omega(n \log n)$ is a lower bound for the expected (w.r.t. coin flips) running time of a Las Vegas algorithm that computes the closest pair. This shows that algorithm of Theorem 4 is still optimal.

Acknowledgements

The authors would like to thank Ian Munro for useful discussions and Kurt Mehlhorn for turning our attention to the tail estimate in [4].

References

[1] M. Ben-Or. *Lower bounds for algebraic decision trees.* Proc. 15th Annual ACM Symp. on Theory of Computing, 1983, pp. 80-86.

[2] J.L. Bentley and M.I. Shamos. *Divide-and-conquer in multidimensional space.* Proc. 8th Annual ACM Symp. on Theory of Computing, 1976, pp. 220-230.

[3] M. Dietzfelbinger and F. Meyer auf der Heide. *A new universal class of hash functions and dynamic hashing in real time.* Proc. ICALP 90, Lecture Notes in Computer Science, Vol. 443, Springer-Verlag, Berlin, 1990, pp. 6-19.

[4] K.E. Clarkson, K. Mehlhorn and R. Seidel. *Four results on randomized incremental constructions.* Report MPI-I-92-112, Max-Planck-Institut für Informatik, Saarbrücken, Germany, March 1992, and Proc. 9th Symp. on Theoretical Aspects of Computer Science (STACS 92), Lecture Notes in Computer Science, Vol. 577, Springer-Verlag, Berlin, 1992, pp. 463-474.

[5] M. Golin, R. Raman, C. Schwarz and M. Smid. *Randomized data structures for the dynamic closest-pair problem.* Proc. 4th ACM-SIAM Symp. on Discrete Algorithms, 1993, pp. 301-310.

[6] S. Khuller and Y. Matias, *A simple randomized sieve algorithm for the closest-pair problem.* Proc. Third Canadian Conf. on Computational Geometry, (1991), pp. 130-134.

[7] H.P. Lenhof and M. Smid. *Enumerating the k closest pairs optimally.* Proc. 33rd Annual IEEE Symp. Foundations of Computer Science, 1992, pp. 380-386.

[8] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, New York, 1985.

[9] M. Rabin, *Probabilistic algorithms,* in "Algorithms and Complexity: New Directions and Recent Results (J.F. Traub ed.)," (1976), pp. 21-39.

[10] J.S. Salowe. *Shallow interdistance selection and interdistance enumeration.* International Journal of Computational Geometry & Applications **2** (1992), pp. 49-59.

[11] C. Schwarz, M. Smid and J. Snoeyink. *An optimal algorithm for the on-line closest pair problem.* Proc. 8th ACM Symp. on Computational Geometry, 1992, pp. 330-336.

[12] R. Seidel. *Small-dimensional linear programming and convex hulls made easy.* Discrete Comput. Geom. **6** (1991), pp. 423-434.

[13] R. Seidel. *Backwards analysis of randomized geometric algorithms* Report TR-92-014, Computer Science Division, University of California Berkeley, Feb. 1992.