# Simple Randomized Algorithms for Closest Pair Problems*

— Extended Abstract —

Mordecai J. Golin† Rajeev Raman‡
Christian Schwarz§ Michiel Smid¶

## 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 appropriately then the lower bound no longer holds. This was first shown by Rabin [8] 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 [8] 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 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 [11, 12].

*This research was supported by the European Community, Esprit Basic Research Action Number 7141 (ALCOM II).

†Hongkong UST, Clear Water Bay, Kowloon, Hongkong. email: golin@cs.ust.hk. Work was done while author was employed at INRIA Rocquencourt, France.

‡UMIACS, University of Maryland, College Park, MD 20742, USA. email: raman@umiacs.umd.edu. Work was done while author was employed at Max-Planck-Institut für Informatik, Germany.

§Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany. email: schwarz@mpi-sb.mpg.de, from Sept. 1993: schwarz@icsi.berkeley.edu.

¶Max-Planck-Institut für Informatik, D-66123 Saarbrücken, Germany. email: michiel@mpi-sb.mpg.de.

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 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. Finally, in Section 4, we mention some extensions of the algorithm. (Details can be found in the full paper.)

## 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 $\delta(S_i)$.

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

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

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.

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.

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. In terms of grids, this 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 $\{\lfloor p_i^{(j)}/d \rfloor, 1 \leq i \leq n, j = 1, 2\}$ 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.

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 $O(i) \cdot 2/(i+1) = O(1)$.)

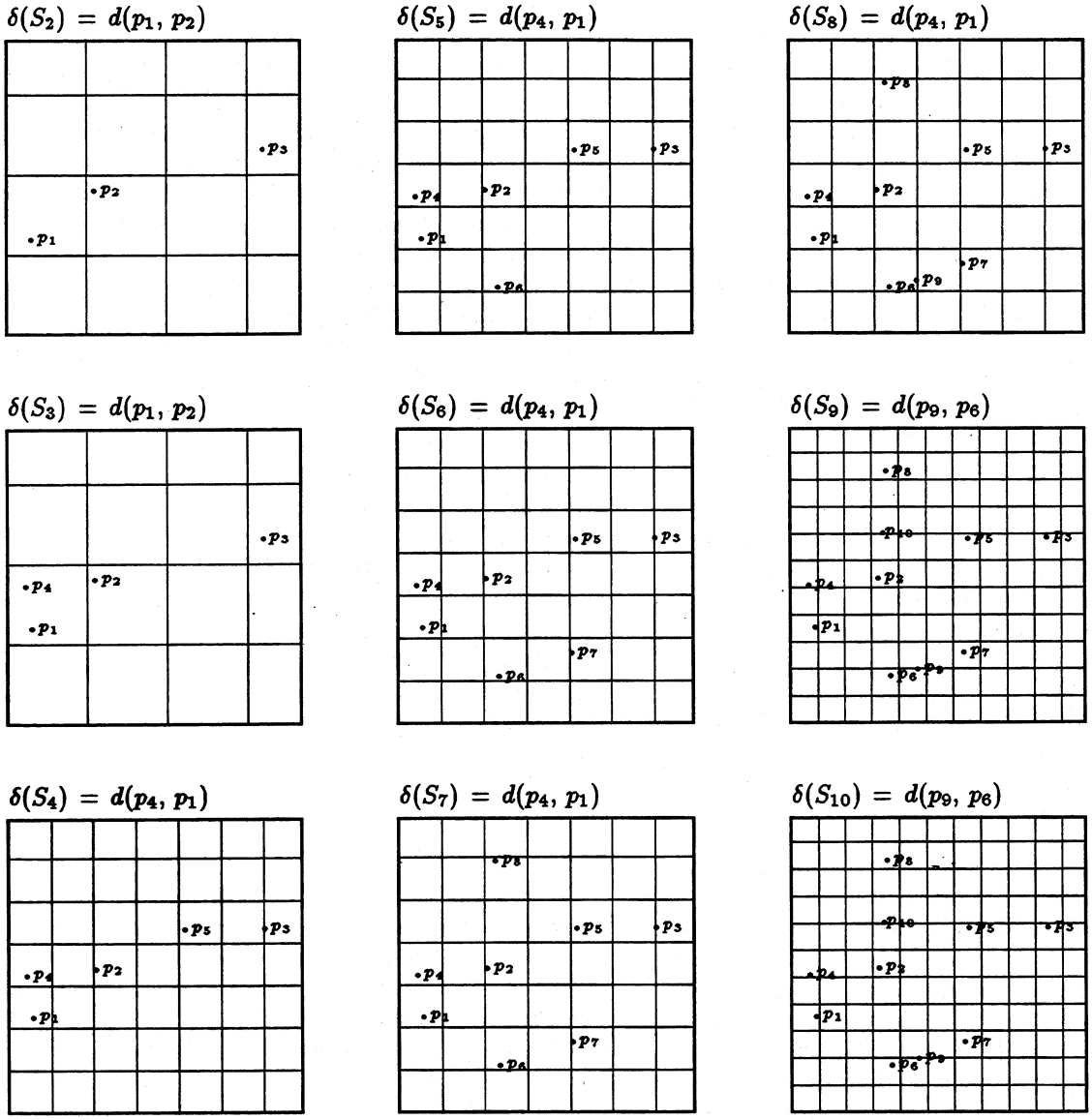


Figure 1: The incremental algorithm running on a set of 10 points. In the beginning the grid has mesh size $d(p_1, p_2)$. Every new minimal distance that is computed during the algorithm causes a refinement of the grid.

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

(1) $\delta := d(p_1, p_2)$; $\mathcal{G} := Build(S_2, \delta)$;
(2) **for** $i := 2$ **to** $n - 1$ **do**
(3) **begin**
(4) $V := \{Report(\mathcal{G}, b) : b$ is a neighbor of the box containing $p_{i+1}\}$;
(5) $d := \min_{q \in V} d(p_{i+1}, q)$;
(6) **if** $d \geq \delta$ **then** $Insert(\mathcal{G}, p_{i+1})$
(7) **else** $\delta := d$; $\mathcal{G} := Build(S_{i+1}, \delta)$;
(8) **end**;
(9) **return**$(\delta)$.

Figure 2: Pseudocode for the closest pair algorithm.

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 3.

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, $1 \leq t \leq \infty$. Suppose $S$ is a collection of $D$-dimensional points, where $D \geq 2$. 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 [10].)

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, 9, 10] 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 $\Pr[\delta(S_{i+1}) < \delta(S_i)] \leq 2/(i+1)$.*

**Proof:** We use Seidel's backwards analysis technique. (See [11, 12].) Consider $S_i$, $p_{i+1}$ and $S_{i+1} = S_i \cup \{p_{i+1}\}$. Let $A := \{p \in S_{i+1} : \exists q \in S_{i+1}$ 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 $\delta(S_{i+1}) < \delta(S_i) \iff p_{i+1} = p$. 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.*
2. *The implementation of the algorithm that uses dynamic perfect hashing finds a closest pair in $S$, in $O(n)$ expected time.*

# 3 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 $c \geq 1$, $\Pr[Y_S \geq cn] \leq e^{2c}/e^2c^c$.*

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: $\Pr[Y_S \geq cn] \leq G_S(a)/a^{cn}$ for any $a \in \mathbb{R}_{\geq 1}$ by Claim 1. By Claim 2, this is at most $\left(\prod_{1\leq i\leq n}\left(1 + \frac{2}{i}(a^i - 1)\right)\right)/a^{cn}$. Applying the inequality $1 + x \leq e^x$ gives $\Pr[Y_S \geq cn] \leq \exp\left(\sum_{1\leq i\leq n}\frac{2}{i}(a^i - 1)\right)/a^{cn}$, which is at most $\exp(2(a^n - 1))/a^{cn}$, since $\frac{2}{i}(a^i - 1) \leq \frac{2}{n}(a^n - 1)$ for each $i \leq n$ and each $a \geq 1$. Choosing $a = c^{1/n}$, we obtain $\Pr[Y_S \geq cn] \leq e^{2c}/e^2c^c$. ∎

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 $O(\log i + cost(p_{i+1}, S_{i+1}))$ and the total work performed by the algorithm is $O(n \log n + Y_S)$.

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

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

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 $1 - O(n^{-t})$ for every $t$. Moreover, a new element can be inserted into a hash table storing $n$ items in time $O(1)$, also with probability $1 - O(n^{-t})$.

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 $1-O(n^{-t/2})$, an insert into the table will take only $O(1)$ time. Moreover, a rebuild on $i$ items will take $O(i)$ time with probability $1 - O(n^{-t/2})$. We can therefore assume that, with probability $1 - O(n^{1-t/2})$, over the entire algorithm, every insert takes $O(1)$ time and every rebuild on $i$ elements takes $O(i)$ time. That is, with probability $1 - O(n^{1-t/2})$, 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, for any positive integer $r$, $Y_S = O(n\log n/\log\log n)$ with probability $1 - O(n^{-r})$. Therefore the algorithm runs in $O(n\log n/\log\log n)$ time with probability $1 - O(n^{1-t/2} + n^{-r})$ for any $t, r \geq 1$. So for any $s \geq 1$, we can choose $t = 2s + 2$ and $r = s$ to obtain the running time with probability $1 - O(n^{-s})$.

We summarize our results:

**Theorem 2** *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* $s$.

# 4 Extensions

We now mention some extensions of the algorithm. (Details can be found in the full paper.) First, 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, 9].) Combined with randomization and the floor function, we get a simple algorithm with expected running time $O(kn)$, which is better than the algorithms in [7, 9] if $k = o(\log n)$.

[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.

Second, all algorithms of this paper assume that the floor function can be computed at unit cost. We have 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. Note that, since the $\Omega(n \log n)$ lower bound still holds for this model, the algorithm is optimal. The main idea of the algorithm is to replace the standard grid, for which the floor function is needed, by a slightly degraded grid, for which we don't need the floor function. The technique that we use appears already in [5, 7].

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

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

[10] 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.

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

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