Nordic Journal of Computing

SIMPLE RANDOMIZED ALGORITHMS FOR CLOSEST PAIR PROBLEMS*

MORDECAI GOLIN
Hongkong UST
Clear Water Bay, Kowloon
Hongkong
golin@cs.ust.hk

RAJEEV RAMAN
Department of Computer Science
King's College London
Strand
London WC2R 2LS, UK
raman@dcs.kcl.ac.uk

CHRISTIAN SCHWARZ
International Computer Science Institute
Berkeley, CA 94704, USA
schwarz@icsi.berkeley.edu

MICHIEL SMID
Max-Planck-Institut für Informatik
D-66123 Saarbrücken, Germany
michiel@mpi-sb.mpg.de.

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. Much of the work reduces to maintaining a dynamic dictionary. Using dynamic perfect hashing to implement the dictionary, the closest pair algorithm runs in $O(n)$ expected time. Alternatively, we can use balanced search trees, and stay within the algebraic computation tree model, which yields an expected running time of $O(n \log n)$. In addition to being quick on the average, the algorithm is reliable: we show that the high-probability bound increases only slightly to $O(n \log n / \log \log n)$ if we use hashing and even remains $O(n \log n)$ if we use trees as our dictionary implementation. Finally, we give some extensions to the fully dynamic closest pair problem, and to the k closest pair problem.

CR Classification: F.2.2

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 studied extensively. 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

* This research was supported by the European Community, Esprit Basic Research Action Number 7141 (ALCOM II). The work was partly done while the first author was employed at INRIA Rocquencourt, France, and visiting Max-Planck-Institut für Informatik, Germany, and the second and the third author were employed at Max-Planck-Institut für Informatik. The first author was also supported by NSF grant CCR-8918152.

Received FILL IN. Revised FILL IN.

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 [16] 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 [10] 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 [16] and [10]. It also differs from them in that it is an *incremental algorithm*. The other two algorithms are inherently static.

With a data structure that delivers the closest pair of the set $S_i := \{p_1, p_2, \ldots, p_i\}$ at hand, the i-th stage of the algorithm inserts p_{i+1} computes the closest pair of the set $S_{i+1} := \{p_1, p_2, \ldots, p_{i+1}\}$. Due to its simplicity, the update method actually has a poor—namely linear—worst case performance. In our algorithm, however, updating is expensive only if the closest pair changes due to the insertion of the new point. This will happen rarely, however, if the input points arrive in *random order*. We will show this using the technique of backwards analysis, due to Seidel. See [21] [22].

So, we analyze the algorithm under the assumption that the input sequence $p_1, p_2, \ldots, p_n$ is a random permutation of the n points, and the running time becomes a random variable. (Note that this variable does not depend on a specific distribution of the input points in D-space.) An algorithm of this kind is called a *randomized incremental algorithm* in the literature, see e.g. [4] [21] [3] [14].

One part of the data structure that has crucial impact on the efficiency of the algorithm is a *dynamic dictionary*. The dictionary implementations that we use are dynamic perfect hashing [7] and balanced search trees. The i-th stage of the algorithm will find the closest pair of the set $S_{i+1} := \{p_1, p_2, \ldots, p_{i+1}\}$ in $O(1)$ expected time if we use hashing or in $O(\log i)$ expected time if we use trees, compared to the $\Omega(i)$ worst case running time for either implementation. This leads to an overall expected running time of $O(n)$ or $O(n \log n)$, respectively.

Assuming that we can generate a random permutation of the input points in linear time, our randomized incremental algorithm gives a very simple solution to the closest pair problem, and its expected running time—in case of the hashing implementation—matches that of [16] [10], in the same model of computation. Apart from that, our algorithm—whatever implementation we take—is also much simpler than the known $O(n \log n)$ deterministic algorithms for finding the closest pair in dimensions higher than two. (See [2] [11] [17] [23] for some of such algorithms.) There exists a simple and elegant sweep line algorithm that runs in time $O(n \log n)$, see [9], but this algorithm only applies to the planar case $D = 2$.

This paper is organized as follows.

In Section 2, we present the randomized incremental algorithm and its analysis. As mentioned above, we have two variants, which depend on the implementation of the underlying dynamic dictionary and run in $O(n)$ or $O(n \log n)$ expected time, respectively.

The high-probability bounds of this algorithm will be investigated in Section 3. The linear expected-time variant actually runs in $O(n \log n / \log\log n)$ time with high probability, while the $O(n \log n)$ expected-time variant also runs in time $O(n \log n)$ with high probability.

Having employed the randomized incremental algorithm to solve the closest pair problem in Sections 2 and 3, we investigate its use as a dynamic algorithm in Section 4. For this purpose, we use a model of *random updates* that has become widely used in the analysis of randomized incremental algorithms. Under this model, we can even support fully-dynamic updates, that is we can efficiently maintain the closest pair under a random sequence of insertions and deletions.

We close the paper with two extensions. First, in Section 5, 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 [11] [17].) Combined with randomization and the floor function, we get a simple algorithm with expected running time $O(n\sqrt{k} \log k)$, which is better than the algorithms in [11] [17] if $k = O((\log n)^{2-\epsilon})$ for any $\epsilon > 0$.

Second, all algorithms of this paper assume that the floor function can be computed at unit cost. In Section 6, 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, and is optimal in this model.

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 $p = (p^x, p^y)$ and $q = (q^x, q^y)$, i.e.,

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

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 $d(p, q) = \delta(S)$.

To begin with, we give a rough description of the data structure that stores the points which are already processed by the incremental algorithm, and sketch the incremental step itself. Let $S_i := \{p_1, p_2, \ldots, p_i\}$ be the set containing the first i points of S. Suppose a square grid with mesh size $\delta(S_i)$ is laid over the plane[1] and each point is stored in the grid box in which it appears.

Now suppose we insert p_{i+1} and want to compute $\delta(S_{i+1})$. The update step is based on the simple observation that $\delta(S_{i+1}) < \delta(S_i)$ holds if and only if there is some point $p \in S_i$ such that $d(p, p_{i+1}) < \delta(S_i)$.

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. See Figure 2.1.

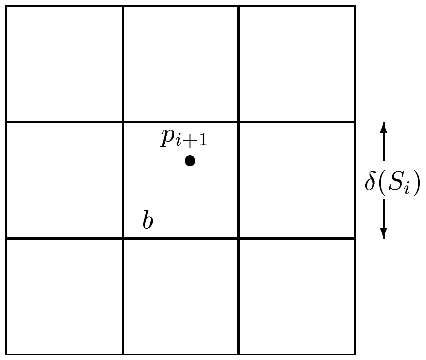

Fig. 2.1: The neighborhood of a grid box b.

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. Suppose the points are fed to the algorithm in the order $p_1, p_2, \ldots, p_n$. 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

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

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 2.2.

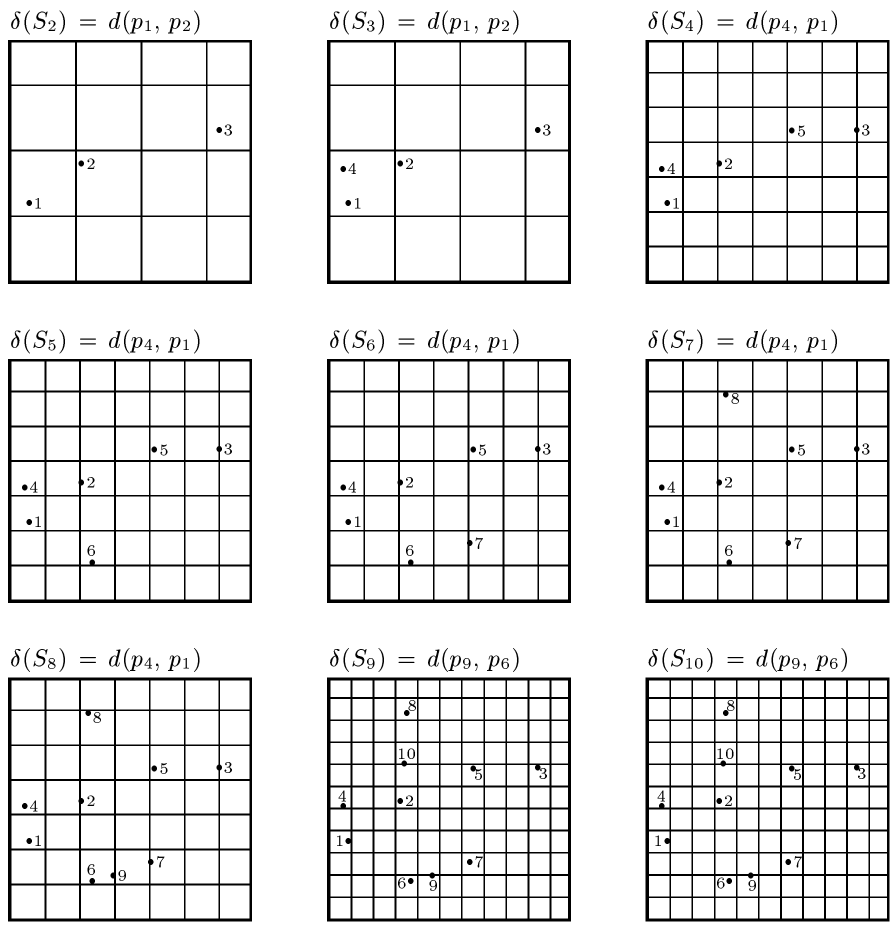

Fig. 2.2: The incremental algorithm running on a set of 10 points. In the beginning the grid has mesh size $d(p_1, p_2)$. Every new minimal distance that is computed during the algorithm causes a refinement of the grid. To avoid clutter, we use i as a shorthand for p_i in the grids.

REMARK 1. 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 $d(q, p_{i+1}) = \delta(S_{i+1})$—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 need the following: let P be a point set, d a positive real number, p a point, $\mathcal{G}$ a grid, and let b denote a grid box. 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 the list containing the points in grid box b.

Pseudocode for the closest pair algorithm using these operations is presented in Figure 2.3.

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

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

Fig. 2.3: Pseudocode for the closest pair algorithm.

How do we actually implement these grid operations? Consider a grid $\mathcal{G}$ with mesh size d, let $p = (p^x, p^y)$ be a point in the plane, and denote the box containing p in $\mathcal{G}$ by b_p. The integer pair $(\lfloor p^x/d \rfloor, \lfloor p^y/d \rfloor)$ is called the *index* of b_p. To store the point set, we maintain a *dictionary*[2] for the non-empty boxes, using their indices as search keys. Moreover, with each box in this dictionary, we store a list of all points that are contained in this box, in arbitrary order. We refer to the whole data structure as the *box dictionary* of the grid.

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 perform a Lookup operation in the box dictionary using this index. If the box is already stored in the dictionary, then we insert the new point into the list that is stored

[2] We use the term dictionary for a data structure that stores a set of items such that the operations Lookup, Insert and Delete can be carried out.

with the box. Otherwise, we insert the new grid box, together with a list containing the new point, into the box dictionary.

What is left open is the implementation of the box dictionary. One standard way is to use balanced binary search trees, which allow query and update operations to be carried out in logarithmic time. To store the box indices, which are integer pairs, in a search tree, we need to have some ordering on them: using the lexicographical ordering on integer pairs will do. So, 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 $Insert(\mathcal{G}, p)$ and $Report(\mathcal{G}, b)$ both cost $O(\log n)$ time. (For a call of *Report*, the length $|b \cap P|$ of the returned list is not charged to *Report* itself, but to the calling operation that will make use of this list.)

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 $\lfloor p^x/d \rfloor$ for each point p. Initialize an empty bucket $B(\lfloor p^x/d \rfloor)$ for all distinct values $\lfloor p^x/d \rfloor$. Moreover, give each point in X a pointer to its bucket. Each bucket corresponds to a vertical slab that contains at least one point of P. 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. Add $(\lfloor p^x/d \rfloor, \lfloor p^y/d \rfloor)$ at the end of the bucket, if it was not stored there yet. 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. Since we can easily maintain the lists X and Y in logarithmic time for each update of the point set, we shall employ this auxiliary data structure whenever we use the tree implementation of the box dictionary. The complexity of *Insert* remains unchanged.

Another way to implement the box dictionary is by using dynamic perfect hashing [7] instead of balanced trees to organize the indices of 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 $O(1)$ deterministic time.

We should point out that dynamic perfect hashing as described by [7] 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 $p = (p^x, p^y)$ in a grid with mesh size d is the integer pair $(\lfloor p^x/d \rfloor, \lfloor p^y/d \rfloor)$. Suppose we know a range $[-x^-, x^+] \times [-y^-, y^+]$, where x^-, x^+, y^-, y^+ are positive real numbers, that contains all the inputs points for our algorithm. We refer to this range as the *frame* in the following. Then, for a grid of mesh size d, the box indices will be in the range $[-\lfloor x^-/d \rfloor \ldots \lfloor x^+/d \rfloor] \times [-\lfloor y^-/d \rfloor \ldots \lfloor y^+/d \rfloor]$. So, when building a grid of mesh size d during the algorithm, we have a bound on the possible indices that will ever occur in the box dictionary *before* we start gridding.

In the static closest pair problem, which we are discussing in this section, all the points are given in advance and so the above frame assumption is trivially fulfilled. (If, however, we want to use the algorithm in a truly dynamic fashion, the need to know a frame that contains the points actually becomes a restricting requirement for the hashing implementation. See Section 4.)

The second requirement for dynamic perfect hashing mentioned above was that we also need to know a prime exceeding the size of the universe. In an actual implementation, we can expect moderate universe sizes due to the limitations to d given by machine precision, and would therefore use a precomputed lookup table to find the desired prime number. In the following paragraph, we discuss how this can be done if such a table is not available, using results of Dietzfelbinger et. al. [5].

Suppose we have to build a dictionary initially containing n keys. We proceed in two steps. First we apply a *universe reduction* to our input keys. This is a function that maps the keys to a smaller range such that the probability of a "collision", i.e. the event that two input keys are mapped to the same reduced key, is very small. In [5], a whole class of possible mappings is given, which allows us to choose the range of the reduced keys. There is a tradeoff between the degree of the compression and the reliability of the reduction. For example, if we want to reduce a set of n keys to a range that is polynomial in n, we can do this with a failure[3] probability that decreases polynomially fast in n. The only precondition that we need for this technique is to know a power of two exceeding all input keys in advance. If we assume that we can compute EXP and LOG in $O(1)$ time, then we can find this number in constant time, by calculating $2^{\lceil \log u \rceil}$, where u is our bound on the keys. Also, computing powers of two is the only non-standard operation coming up in the universe reduction mapping itself, so using the above assumption the mapping can be evaluated in constant time for each input key. Now suppose we have reduced a set of n keys to a polynomially-bounded range. Then, another result in [5] allows us to compute a prime exceeding this range in time $O(\log^4 n)$, again with polynomial failure probability, as for the previously applied universe reduction. This running time is dominated by the (linear) time that is needed anyway to actually build the dictionary containing n items. During the operation of the dynamic dictionary, we also apply the universe reduction mapping first, and then use the reduced key in the hashing scheme provided by [7]. Further details on the application of these hashing techniques to our gridding algorithm are beyond the scope of this article. They are covered in detail in [19]. Let us only remark that the additional effort made in each dictionary operation in order to fulfill the aforementioned requirements of dynamic perfect hashing does not affect the complexities for our grid operations stated above, namely $O(1)$ deterministic time for *Report*, $O(1)$ expected time for *Insert* and $O(n)$ expected time for *Build*.

[3] We actually reapply the whole procedure until we get the desired behavior, so the term "failure" only denotes the fact that we need more than one trial to achieve our goal.

We now analyze the cost of the closest pair algorithm. Let $Q(n)$, $U(n)$ and $P(n)$ be the time for *Report*, *Insert* and *Build* for a data structure of size n, respectively. Refer to Figure 2.3.

Let i be fixed. The update step to compute $\delta(S_{i+1})$ upon insertion of p_{i+1} includes lines 4-7. Line 4 of the algorithm calls *Report* 9 times to find at most 36 points. Therefore lines 4-5 use $O(1 + Q(i)) = O(Q(i))$ time. Lines 6 and 7 take time $U(i)$ and $P(i + 1)$, respectively. Note that line 7 is called if and only if $\delta(S_{i+1}) < \delta(S_i)$. We capture this with the following definition.

DEFINITION 1. *For any set V of points and any point $p \in V$, define*

$$X(p, V) = \begin{cases} 1 & \textit{if } \delta(V) < \delta(V \setminus \{p\}) \\ 0 & \textit{otherwise.} \end{cases}$$

Denoting the running time for the i-th stage of the algorithm by T_i, we have

$$T_i = O(Q(i) + U(i) + X(p_{i+1}, S_{i+1}) \cdot P(i + 1)). \tag{2.1}$$

An input sequence that causes the closest pair to change at each stage will lead to quadratic running time, since $P(i + 1) = \Omega(i)$ for both of our anticipated dictionary implementations.

Let us now analyze the case that the points are 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. Then, for each stage i, $X(p_{i+1}, S_{i+1})$ is a random variable and by Definition 1,

$$\mathrm{E}[X(p_{i+1}, S_{i+1})] = \Pr[\delta(S_{i+1}) < \delta(S_i)].$$

We will prove in Lemma 1 below that this probability is at most $2/(i + 1)$, and thus we get the following for the expected running time of the i-th stage:

$$\mathrm{E}[T_i] = O(\mathrm{E}[Q(i)] + \mathrm{E}[U(i)] + 2/(i + 1) \cdot \mathrm{E}[P(i + 1)]). \tag{2.2}$$

Note that for the hashing-based implementation, the random variable $P(i + 1)$ depends on the random choices made by the hashing algorithm. These random choices have nothing to do with the variable $X(p_{i+1}, S_{i+1})$. Therefore, the two random variables are independent and we can multiply their expectations and get $\mathrm{E}[X(p_{i+1}, S_{i+1}) \cdot P(i+1)] = \mathrm{E}[X(p_{i+1}, S_{i+1})] \cdot \mathrm{E}[P(i+1)] = 2/(i + 1) \cdot \mathrm{E}[P(i + 1)]$.

We can now fill in the running times of the basic grid operations for each of the two implementations. Using trees, *Insert* and *Report* take logarithmic time, and *Build* runs in linear time, i.e. $Q(n) = U(n) = O(\log n)$ and $P(n) = O(n)$ for a set of n points. All running times are deterministic. Thus, the i-th stage takes $O(\log i)$ deterministic time plus $O(1)$ expected time in the tree implementation. This separation of deterministic time and expected time will be crucial for the high-probability running time of the algorithm, to be discussed in Section 3.

Using dynamic perfect hashing, *Report* takes $O(1)$ deterministic time, *Insert* takes $O(1)$ expected time, and *Build* takes $O(n)$ expected time. Therefore we have $E[T_i] = O(1)$.

In summary, 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.

It remains to prove Lemma 1.

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 [21] [22].) 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

$$\delta(S_{i+1}) < \delta(S_i) \text{ if and only if } p_{i+1} \in A.$$

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

$$\delta(S_{i+1}) < \delta(S_i) \text{ if and only if } p_{i+1} = p.$$

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 $2/(i+1)$. $\square$

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 [23].)

We summarize our result:

THEOREM 1. *Let S be a set of n points in D-space.*

(1) Using a balanced search tree to implement the box dictionary, the algorithm of Figure 2.3 finds a closest pair in S in $O(n \log n)$ expected time.

(2) Using dynamic perfect hashing to implement the box dictionary, the algorithm of Figure 2.3 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 [3] for obtaining tail estimates on the space complexity of some randomized incremental constructions. While we only needed expected-time bounds for dynamic perfect hashing in the previous section, we shall now use the stronger high-probability bounds that were actually proven in the paper of Dietzfelbinger and Meyer auf der Heide [7].

In each iteration of the closest pair algorithm of Figure 2.3, 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. From Equation 2.1, the rebuilding cost of the i-th stage of the algorithm is $X(p_{i+1}, S_{i+1}) \cdot P(i+1)$, for $2 \leq i \leq n-1$. The sum of these variables describes the overall rebuilding cost of the algorithm. The expected value of this variable was studied in the previous section. Now we are aiming at a tail estimate.

According to Definition 1, we define, for any set V and any point $p \in V$,

$$cost(p, V) := X(p, V) \cdot |V| = \begin{cases} |V| & \text{if } \delta(V) < \delta(V \setminus \{p\}) \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

That is, if we already have inserted the points of $V \setminus \{p\}$, then $cost(p, V)$ expresses the rebuilding cost of the closest pair algorithm when inserting p.

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

$$Y_S := \sum_{i=3}^{n} cost(p_i, S_i).$$

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 [3]. 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 Y be a non-negative random variable that takes only integer values. The *probability generating function (pgf)* of Y is defined by

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

CLAIM 1. *For any $h \geq 0$ and $a \geq 1$*

$$\Pr[Y \geq h] \leq G_Y(a)/a^h.$$

PROOF.

$$\begin{aligned} G_Y(a) &= \sum_{j \geq 0} \Pr[Y = j] \cdot a^j \\ &\geq \sum_{j \geq h} \Pr[Y = j] \cdot a^j \\ &\geq a^h \sum_{j \geq h} \Pr[Y = j]. \end{aligned}$$

□

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

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 $x \geq 1$, and for all sets S, $|S| = n$,*

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

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

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

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,

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

□

Proof of Lemma 2: We apply the above claims:

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

□

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 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 regrids 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 any 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. Instead of the expected-time bounds used in the previous section, Dietzfelbinger and Meyer auf der Heide actually proved the following stronger result:

LEMMA 3. ([7]) *Let $U = [0\ldots u-1]$, u prime, be the universe, and assume we perform at most n update operations on an initially empty set S. Then there is a data structure using $O(n)$ space that implements a dictionary with the following properties:*

(i) *Each* LOOKUP *takes* $O(1)$ *time in the worst case,*

(ii) *each of the* n *update operations* INSERT *and* DELETE *takes* $O(1)$ *time in the worst case,*

and the probability that (ii) is not fulfilled is $O(n^{-t})$ *for any fixed positive integer* t. *The setup time for the data structure is constant.*

So, in particular, the dictionary guarantees this bound on the failure probability for each single update operation. Thus, for any $1 \leq i \leq n$, we can build a dictionary that initially contains i items in $O(i)$ time with n-polynomial failure probability, such that each of the following up to $n - i$ update operations takes $O(1)$ time with n-polynomial failure probability. We overcome the requirements of dynamic perfect hashing stated in the lemma as we did in the previous section. Specifically, if we do not have access to a prime exceeding the size of the universe, then we can find one in time $O(\log^4 n)$.

To sum up, we obtain the following running times for our basic grid operations. Let $1 \leq i \leq n$ be fixed. The time for *Report*, denoted by $Q(i)$, is $O(1)$ in the worst case, and the times for *Build* and *Insert*, denoted by $P(i)$ and $U(i)$, respectively, are $P(i) = O(i + \log^4 n)$ and $U(i) = O(1)$, each with probability $1 - O(n^{-t})$ for any fixed integer t.

To determine the running time for the whole closest pair algorithm, we simply add the failure probabilities occurring in each single stage. Note that the work for all rebuildings in the first $\log^4 n$ stages is bounded by $O(\log^8 n)$, with a failure probability of at most $O(\log^4 n \cdot n^{-t}) = O(n^{1-t})$. Therefore, the total work performed by the algorithm is $O(n + \sum_i cost(p_i, S_i)) = O(n + Y_S)$ with probability $1 - O(n^{1-t})$.

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 $1 - O(n^{1-t} + n^{-r})$ for any $t, r \geq 1$. So for any $s \geq 1$, we can choose $t = s + 1$ and $r = s$ to obtain the running time with probability $1 - O(n^{-s})$.

We summarize our results:

THEOREM 2. *Let* S *be a set of* n *points in* D*-space. The tree-based implementation of the closest pair algorithm runs in* $O(n \log n)$ *time with probability* $1 - O(n^{-s})$ *for any* 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 any* s.

4. A fully-dynamic algorithm

From the previous two sections, we see that the randomized incremental algorithm is a dynamic algorithm by its design, if we do not have to take care for creating a random permutation of the input points at the beginning. Furthermore, our grid data structure can handle deletions of points in a way completely analogous to insertions.

The deletion algorithm

To obtain a deletion algorithm for the data structure as a whole, it remains to discuss how to maintain the closest pair.

The minimal distance in the set *increases* if a closest pair point is deleted and there is no other pair that attains the minimal distance. (We can check this by maintaining a list of all closest pairs, as described in Section 2.) If the minimal distance remains unchanged, the point is deleted from the grid and the closest pair list is updated, if necessary. If, however, the minimal distance increases due to the deletion, then we cannot use the data structure to find a new closest pair, as we were able to in the insertion case (Figure 2.1). Rather, the algorithm is forced to recompute the closest pair from scratch. After that, we can, as in the insertion case, build a new grid data structure whose mesh size is the minimal distance.

REMARK 2. It is natural to re-run the randomized incremental algorithm itself to compute the closest pair from scratch. In this case, the latter step of building the new grid data structure is obsolete, because the grid is already built during the recomputation of the closest pair.

With an analysis similar to the one in Section 2, using the same notation, we get that the running time of the deletion algorithm is

$$O\left(Q(n) + U(n) + X(p,S) \cdot P(n)\right)$$

where $X(p,S)$ is as defined in Definition 1. While $Q(n)$ and $U(n)$ are the same as in Section 2, $P(n)$ now additionally includes the running time of a static closest pair algorithm on a set of n points in case of a deletion.

Using the randomized incremental algorithm of Section 2 itself for this purpose, Theorem 1 yields $P(n) = O(n)$ expected for the hashing-based implementation and $P(n) = O(n \log n)$ expected for the tree-based implementation. This is optimal as long as expected running times are concerned.

Using the running times for the grid operations as stated in Section 2, we obtain

LEMMA 4. *Let S be a set of n points in D-space.*

(1) In the hashing-based implementation, we can maintain the closest pair in S under insertions and deletions of points such that an insertion takes $O(1 + X(p, S \cup \{p\}) \cdot n)$ expected time and a deletion takes $O(1 + X(p,S) \cdot n)$ expected time, under the assumption that the algorithm knows in advance a frame containing all points that will ever be inserted.

(2) In the tree-based implementation, we can maintain the closest pair in S under insertions and deletions of points such that an insertion takes $O(\log n + X(p, S \cup \{p\}) \cdot n)$, deterministic time and a deletion takes $O(\log n + X(p,S) \cdot n \log n)$ expected time.

REMARK 3. Note that $X(p, S)$ is not yet considered as a random variable at the moment. The expectations in the time bounds of Lemma 4 come from the expected values of $U(n)$ and $P(n)$. Also, the frame assumption discussed in Section 2 has to be made since we are no longer assuming to know all input points in advance.

Random updates

The randomized incremental algorithm in Section 2 was analyzed under the assumption that the input points are in random order. For this purpose, we generated a random permutation of the inputs points at the beginning. If we use the algorithm in a dynamic fashion, we make assumptions on the order in which the input is processed and call an update sequence that fulfills these assumptions a random one. We treat insertions and deletions. The assumptions that we make are the ones that are usually made in the literature, see e.g. [12] [13] [18] [8] [14]. For a detailed description, see [14, pp. 126-128].

Let S be the set of all points that are involved in an update operation. Suppose we perform a sequence of n updates. We denote the point involved in the i-th update by p_i, the point set after the i-th update by S_i, and the size of S_i by m_i.

The sequence is called a *random update sequence* if

(1) Each $p \in S$ is equally likely to be p_i.

(2) Each S_i is a random subset of S of size m_i.

REMARK 4. Note that, for the special case when only insertions are performed, this just means that all permutations of S are equally likely. This was the assumption under which we analyzed the insertion algorithm in Section 2.

By symmetry, we have

FACT 1. Let S be a random update sequence, with p_i, S_i, $0 \leq i \leq n$, as defined above.

(1) If the i-th update is an insertion, then p_i is a random point of $S_i = S_{i-1} \cup \{p_i\}$,

(2) If the i-th update is a deletion, then p_i is a random point of $S_{i-1} = S_i \cup \{p_i\}$.

This fact gives us the conditions that we needed for Lemma 1. It follows that

$$\mathrm{E}[X(p_i, S_i)] = \Pr[\delta(S_i) < \delta(S_{i-1})] \leq 2/m_i, \tag{4.1}$$

for an insertion, and

$$\mathrm{E}[X(p_i, S_i \cup \{p_i\})] = \Pr[\delta(S_{i-1}) < \delta(S_i)] \leq 2/(m_i + 1) \leq 2/m_i, \qquad (4.2)$$

for a deletion.

We can now analyze the expected running times of Lemma 4 with the *random variables* $X(p_i, S_i)$ and $X(p_i, S_i \cup \{p_i\})$ given in (4.1) and (4.2), respectively. The running time of an insertion is dominated by that of a deletion, which is proportional to

$$1 + \mathrm{E}[X(p_i, S_i \cup \{p_i\}) \cdot P(m_i)] \leq 1 + 2/m_i \cdot O(m_i) = O(1)$$

for the hashing-based implementation. Note that, as in the analysis of the i-th stage of the incremental algorithm in Section 2, the random variables $X(p_i, S_i \cup \{p\})$ and $P(m_i)$ are independent and we can thus multiply their expectations.

Similarly, the expected running time of a deletion in the tree-based implementation is proportional to $\log m_i + 2/m_i \cdot O(m_i \log m_i) = O(\log m_i)$.

THEOREM 3. *Let S be the set of points that is involved in a sequence of n fully-dynamic random updates, and let R be a random subset of S of size m. Consider a random update on R.*

(1) The hashing-based implementation maintains the closest pair of R in $O(1)$ expected time, under the assumption that a frame containing all the elements of S is known in advance.

(2) The tree-based implementation maintains the closest pair of R in expected time $O(\log m)$.

REMARK 5. Note that we talk about an update sequence only for the matter of analysis. The algorithm is dynamic, it does not know anything about S—except the mild restriction that we need to know a frame containing all the points of S for the hashing based implementation—and it does not need to know the number of updates performed either.

High probability bounds

We can extend the tail estimate for the randomized incremental algorithm given in Section 3 to a sequence of fully-dynamic random updates. This is achieved by a simple modification of the proof in Section 3. In the case where only insertions are performed, we have $|S_i| = i$ for all i, where i is the size of the point set after the i-th update operation. We essentially rewrite the proof with $m_i := |S_i|$ denoting the size of the set after the i-th update instead of i.

Since in general $m_i \ll i$, it is not so useful to give bounds for a sequence of updates in terms of the total number of operations. Rather, we consider a random subset $R \subseteq S$, whose size is conveniently denoted by n, and look at a sequence of $n/2$ random updates $p_1, \ldots, p_{n/2}$ that are performed, starting

	implementation of the grid structure	
	tree	hashing
[16]	$n \log n$	$n \log n / \log \log n$
Theorem 2	$n \log^2 n / \log \log n$ *	$n(\log n / \log \log n)^2$

TABLE I: High probability bounds of the fully-dynamic algorithm for the time to execute a sequence of $n/2$ random updates starting with a set of size n. The tree-based variants do not need to know a frame containing all the points in advance, and the algorithm $*$ can be modified to fit in the algebraic decision tree model, using a technique that will be explained in Section 6.

with R. Then, all the set sizes occurring in the sequence are of the same order of magnitude.

We omit the details, which can be found in [19], and only give the results. The running times depend on the algorithm that is used to recompute the closest pair when this is needed after the deletion of a point. We show two variants. One uses Rabin's algorithm, which still runs in linear time with very high, in fact exponential, probability. The other variant uses the randomized incremental algorithm itself. See Table I.

5. The k closest pairs problem

In this section we describe how to modify the closest pair algorithm of Section 2 to provide a simple solution for 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. For $i \geq 1$, let $S_i = \{p_1, p_2, \ldots, p_i\}$. Our algorithm incrementally calculates $\delta^k(S_i)$. 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, called the D-tree, which contains k pairs of points from S_i that are at most $\delta^k(S_i)$ apart, sorted in increasing order by distance.

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 having distance less than $\delta^k(S_i)$ to 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 pairs of 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 Section 2 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 D-tree.

For each inserted distance, we delete the maximal element stored in the D-tree. In this way, the D-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 Section 2 to establish Lemma 1 shows that—if the points are fed to the algorithm in a random order—the probability of the k-th smallest distance changing in the i-th stage is at most $2k/(i+1)$.

Therefore, the expected cost of inserting the $(i+1)$-st point is $O(k+\log i)$ if trees are used and $O(k)$ if dynamic perfect hashing is used.

We can speed up the running time of an insertion in the amortized sense, by the following simple modification. The data structure no longer has a grid size $\delta = \delta^k(S_i)$ at the i-th stage, but one that satisfies $\delta^k(S_i) \leq \delta \leq \delta^{2k}(S_i)$. Also, we store the ordered sequence of the ℓ smallest distances in the point set S_i in the D-tree, where ℓ may vary between k and $2k$. None of these distances is greater than δ. Additionally, we maintain a pointer to the item stored at the k-th position. In this way, we can still identify the k-th smallest distances in the set.

The algorithm to insert p_{i+1} is as follows. First, it finds all points having distance less than δ of p_{i+1}, exactly as before. Since $\delta \leq \delta^{2k}(S_i)$, the number of points found increases only by a constant factor compared to the method described before, i.e. it is still $O(\sqrt{k})$. For each of these points, we calculate its distance to p_{i+1} and insert it into the D-tree. Contrary to the original method, we do not delete the maximal distance stored in the D-tree in exchange for this newly inserted distance. Rather, we move the pointer indicating the k-th position if necessary. Having processed all distances between p_{i+1} and the points in its neighboring boxes, we know $\delta^k(S_{i+1})$ and the k closest pairs of S_{i+1}. The time used for this part of the insertion algorithm is still $O(\sqrt{k}\log k)$.

If $l \leq 2k$, we finish the operation by inserting p_{i+1} into the current grid, at a cost of $O(\log i)$ in the tree-based implementation or $O(1)$ in the hashing-based implementation, respectively. Otherwise, we discard the current grid and insert all points of S_{i+1} into a new grid with mesh size $\delta^k(S_{i+1})$. Also, only the k smallest distances remain in the tree. This can be achieved in time $O(\log k)$ by splitting the tree at the k-th position. (Recall that we maintain a pointer to this position during the algorithm.) This computation is dominated by the $\sqrt{k}\log k$ term that comes up for every insertion, not only for those that build a new grid. Thus, we need $O(i)$ extra time for the insertion of p_{i+1} if a new grid is built.

Since k insertions into the D-tree are performed between two successive rebuildings, the amortized cost of regridding is $O(i/k)$ for each distance inserted into the D-tree. It remains to bound the expected number of insertions into the D-tree for the i-th stage of the algorithm. New values are brought into the D-tree only if there are points $p \in S_i$ with $d(p, p_{i+1}) < \delta$. Note that $\delta = \delta^{\ell}(S_i)$, where ℓ is the number of distances in the D-tree before the insertion of p_{i+1}, and that δ is the unique maximal distance stored in the tree, i.e. $\delta^{\ell-1}(S_i) < \delta^{\ell}(S_i)$. Hence, $\delta^{\ell}(S_{i+1}) < \delta^{\ell}(S_i)$ if and only if new distances enter the D-tree in the i-th stage. Since $\ell \leq 2k$, the probability that the D-tree is modified in the i-th stage is bounded by $4k/(i+1)$. Furthermore, since at most $O(\sqrt{k})$ distances can be inserted into the D-tree in a single stage, the expected number of such distances is $O(k\sqrt{k}/i)$ in the i-th stage. We have seen above that the amortized rebuilding cost is $O(i/k)$ for each new distance in the D-tree, so the expected amortized rebuilding cost of the i-th stage is $O(\sqrt{k})$.

To sum up, the expected amortized running time for the whole algorithm to insert p_{i+1} is $O(\log i + \sqrt{k}\log k)$ in the tree-based implementation and $O(\sqrt{k}\log k)$ in the hashing-based implementation. Thus, the expected time to find $\delta^k(S) = \delta^k(S_n)$ is $O(n(\sqrt{k}\log k + \log n))$ or $O(n\sqrt{k}\log k)$, respectively.

THEOREM 4. *The tree-based implementation of the k closest pairs algorithm runs in $O(n(\sqrt{k}\log k + \log n))$ expected time, whereas the hashing-based implementation runs in $O(n\sqrt{k}\log k)$ expected time.*

6. 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 $O(n)$ algorithm. (See [15].)

In this section, we sketch how to obtain an $O(n \log n)$ expected time algorithm for the closest pair problem that fits in the algebraic decision tree model. The data structure that we use is called "degraded grid". Originally due to Lenhof and Smid [11], this method was significantly simplified by Datta et. al. [6]. Their degraded grid looks very much like a standard grid, has basically the same properties, but one can build it and search in it without using the floor function. Here, we slightly modify and dynamize that method.

As in the previous sections, we restrict ourselves to the planar case for simplicity. The method also works for arbitrary dimension D, however. To give an intuitive idea, consider a standard grid of mesh size d, or d-grid for short. There, we divide the plane into slabs of width d and fix $(0,0)$ as a lattice point. This suffices to specify the grid completely and gives rise to an *implicit* storage of the slabs: for any point p in the plane, we can identify the vertical or horizontal slab containing it by computing $\lfloor p^x/d \rfloor$ or $\lfloor p^y/d \rfloor$, respectively.

A degraded grid depends on the point set, say S, that is being stored. We still divide the plane into horizontal and vertical slabs. To avoid the use of the floor function, we store these slabs *explicitly*, by keeping a dictionary for the coordinates of the endpoints. We call these dictionaries the *slab dictionaries*. We implement a slab dictionary using a balanced search tree. The endpoints of slabs are not necessarily multiples of d, as it was the case for the standard grid. Also, the width of a slab need not be exactly d. In a *degraded d-grid storing S*, we maintain the following invariant:

(1) Each slab has width at least d.

(2) Each slab containing a point of S has width at most $2d$.

(3) There are no two empty slabs that are adjacent.

Note that by the third condition, the number of slabs per dimension is at most $2n$, where n is the number of points being stored. Thus, the degraded grid only needs linear storage. Figure 6.1 shows an example.

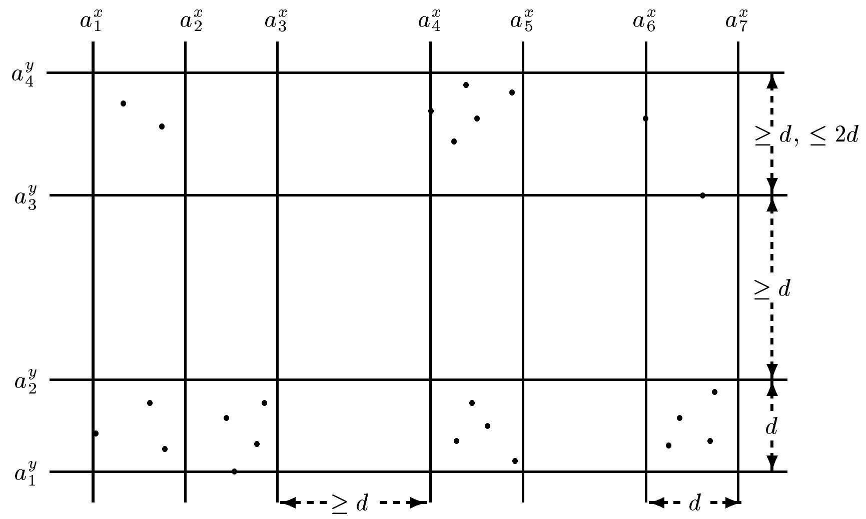

Fig. 6.1: Example of a degraded d-grid. It is dependent on the point set stored in it.

Now, if we are given point p, and want to find out the box containing it in the degraded grid, we do this by determining the horizontal and vertical slab containing it. We only need to locate each coordinate of p in the corresponding slab dictionary. This takes $O(\log n)$ time.

We modify a degraded d-grid for S upon insertions or deletions of points, as follows. Suppose we want to insert a new point p. If the slabs containing p have width at most $2d$, then nothing needs to be done. Otherwise, assume w.l.o.g. that the vertical slab containing p has a width exceeding $2d$. According to the invariant, this slab was previously empty, and it now needs to be partitioned such that each of the resulting slabs has width at least d and the slab containing p has width at most $2d$. We can achieve this because the width of the slab which is partitioned is greater than $2d$. The partition of the slab is implemented by inserting one or two new slab boundaries into the corresponding slab dictionary.

Maintaining the degraded grid upon deletion of a point p from S is analogous. If a slab becomes empty due to the deletion of a point p, and if one of the neighboring slabs is also empty, then we remove garbage from the slab dictionary by combining these two intervals to one, thereby reestablishing item 3 of our invariant. Thus, the degraded grid can be maintained under insertions and deletions in $O(\log n)$ time.

Also note that the slab dictionaries can be built in linear time if sorted lists X and Y containing the x- and y-coordinates of the points, respectively, are available: the sorted lists of slab endpoints can be obtained by a linear scan through the lists X and Y. Moreover, given the sorted lists of slab endpoints, we can construct a lexicographically ordered list of non-empty boxes in linear time, in a way that is similar to the procedure described for grids in Section 2. The buckets correspond to the non-empty vertical slabs. Consider the walk through the list Y, where we distribute the box indices of the points into the buckets. We maintain a pointer to the currently last horizontal slab that has been reached. If a new point falls outside this slab, then it must be in a slab with a larger y-coordinate, and we simply scan upwards through the sorted list of slabs until we have found the slab containing the point. Thus, we walk through this list only once during the whole construction, and the time spent is still linear.

Let us now turn to the closest pair algorithm and the grid operations needed for it. We consider the tree implementation for the box dictionary, and to store the set $S_i = p_1, p_2, \ldots, p_i$, we replace the standard grid of mesh size $\delta(S_i)$ by a degraded $\delta(S_i)$-grid. To run the algorithm as shown in Figure 2.3, we need to extend the notion of neighborhood to degraded grids. Note that the degraded grid retains the standard grid property that each box b is adjacent to exactly 9 boxes, where b is considered to be adjacent to itself. Thus, we do not need to change the notion of neighborhood for the degraded grid. Also note that since the side length of each box is at least $\delta(S_i)$ in a degraded $\delta(S_i)$-grid, the property that all points within distance $\delta(S_i)$ of p_{i+1} can be found by checking the boxes in the neighborhood of the box containing p_{i+1} is preserved, which establishes the correctness of the algorithm.

The analysis of the closest pair algorithm remains the same as before, except that we now need logarithmic time to identify the box containing a given point, and to modify the degraded grid upon insertion or deletion of a point. However, having identified the box containing a point, this box must be found in the box dictionary in operation *Report*. Likewise, the grid operations *Insert* and *Delete* may need to modify the box dictionary. Since search and update operations on the box dictionary already take logarithmic time in the tree implementation, the aforementioned additional computations needed for the degraded grid do not affect the complexities of the grid operations.

Finally, the side length of a non-empty box in a degraded $\delta(S_i)$-grid is at most $2\delta(S_i)$, which means that the number of points returned by operation *Report* is still constant.

THEOREM 5. *Let S be a set of n points in D-space. The degraded grid implementation of the randomized incremental algorithm 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, and is therefore optimal. Moreover, this algorithm runs in $O(n \log n)$ time with probability $O(n^{-s})$ for any s.*

PROOF. The claimed time bounds follow from the results of Sections 2 and 3 and the discussion above. It remains to show that the algorithm is optimal.

Note that the algorithm is of 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 the algorithm of Theorem 5 is still optimal. □

7. Conclusion

In this paper, we have presented a simple data structure to maintain the closest pair in a set of points. With this structure, we can find the new closest pair after the insertion of a point efficiently. This is not the case for deletions: the minimal distance has to be recomputed from scratch, if it is changed by the deletion. However, both insertions *and* deletions require a rebuilding of the data structure when the closest pair changes. The extreme dependence of the running time on a change of the closest pair leads to the model of *random updates.* In this model, the probability of a closest pair change is very small, which leads to good results. Given all points at the start of the computation, we can guarantee random updates by generating a random permutation of the inputs.

If we use the algorithm in a truly dynamic fashion, the results may be interpreted as the algorithm's behavior on a typical update sequence. Even more interesting from this point of view are high probability bounds. They show that the algorithm performs well except for very rare occasions. See Mulmuley [14, pp. 127-128] for a discussion of this issue.

Again, it is important that the randomization is solely with respect to the order of the updates and has nothing to do with the positions of the points in the input space.

Many algorithms using the *randomized incremental construction* paradigm are substantially simpler than their deterministic counterparts, as described in [12] [13] [18] [8] [14]. The algorithms in this paper are another strong example for this phenomenon. This leads to the question whether one can transform the *conceptual* simplicity into a practically efficient algorithm that is competitive with the methods that have been implemented previously. This issue is addressed in a current project [20], where several static and dynamic closest pair algorithms are implemented and compared under various input distributions.

Acknowledgements

The authors would like to thank Ian Munro for useful discussions and Kurt Mehlhorn for turning our attention to the tail estimate in [3], and Torben Hagerup for pointing us to [5].

References

[1] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. 15th Annu. ACM Sympos. Theory Comput.*, pages 80–86, 1983.

[2] J. L. Bentley and M. I. Shamos. Divide-and-conquer in multidimensional space. In *Proc. 8th Annu. ACM Sympos. Theory Comput.*, pages 220–230, 1976.

[3] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. In *Proc. 9th Sympos. Theoret. Asp. Comp. Sci.*, volume 577 of *Lecture Notes in Computer Science*, pages 463–472. Springer-Verlag, 1992.

[4] K. L. Clarkson and P. W. Shor. Algorithms for diametral pairs and convex hulls that are optimal, randomized, and incremental. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 12–17, 1988.

[5] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. Unpublished manuscript, November 1992.

[6] A. Datta, H.-P. Lenhof, C. Schwarz, and M. Smid. Static and dynamic algorithms for k-point clustering problems. In *Proc. 3rd Workshop on Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 265–276. Springer-Verlag, 1993.

[7] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proc. 17th Internat. Colloq. Autom. Lang. Prog.*, volume 443 of *Lecture Notes in Computer Science*, pages 6–19. Springer-Verlag, 1990.

[8] O. Devillers, S. Meiser, and M. Teillaud. Fully dynamic Delaunay triangulation in logarithmic expected time per operation. *Comput. Geom. Theory Appl.*, 2(2):55–80, 1992.

[9] K. Hinrichs, J. Nievergelt, and P. Schorn. Plane-sweep solves the closest pair problem elegantly. *Information Processing Letters*, 26:255–261, 1988.

[10] S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. In *Proc. 3rd Canad. Conf. Comput. Geom.*, pages 130–134, 1991.

[11] H.-P. Lenhof and M. Smid. Enumerating the k closest pairs optimally. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 380–386, 1992.

[12] K. Mulmuley. Randomized multidimensional search trees: dynamic sampling. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 121–131, 1991.

[13] K. Mulmuley. Randomized multidimensional search trees: Lazy balancing and dynamic shuffling. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 180–194, 1991.

[14] K. Mulmuley. *Computational Geometry – An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[15] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction.* Springer-Verlag, New York Berlin Heidelberg Tokyo, second edition, 1988.

[16] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity*, pages 21–30. Academic Press, New York, NY, 1976.

[17] J. S. Salowe. Enumerating interdistances in space. *Internat. J. Comput. Geom. Appl.*, 2:49–59, 1992.

[18] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *Proc. 32nd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 197–206, 1991.

[19] C. Schwarz. *Data structures and algorithms for the dynamic closest pair problem.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1993.

[20] C. Schwarz. Dynamic closest pair algorithms: implementation and application. Unpublished manuscript, 1994.

[21] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete Comput. Geom.*, 6:423–434, 1991.

[22] R. Seidel. Backwards analysis of randomized geometric algorithms. Technical Report TR-92-014, Dept. Comput. Sci., Univ. of California Berkeley, Berkeley, CA, 1992.

[23] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12:18–29, 1994.