Chapter 33
Randomized Data Structures for the Dynamic Closest-Pair Problem

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

Abstract

We describe a new randomized data structure, the *sparse partition*, for solving the dynamic closest-pair problem. Using this data structure the closest pair of a set of n points in k-dimensional space, for any fixed k, can be found in constant time. If the points are chosen from a finite universe, and if the floor function is available at unit-cost, then the data structure supports insertions into and deletions from the set in expected $O(\log n)$ time and requires expected $O(n)$ space. Here, it is assumed that the updates are chosen by an adversary who does not know the random choices made by the data structure. The data structure can be modified to run in $O(\log^2 n)$ expected time per update in the algebraic decision tree model of computation. Even this version is more efficient than the currently best known deterministic algorithms for solving the problem for $k > 1$.

1 Introduction

We consider the *dynamic closest-pair problem*: We are given a set S of points in k-dimensional space (we assume k is an arbitrary constant) and want to keep track of the closest pair of points in S as S is being modified by insertions and deletions. Distances are measured in the L_t-metric, where t is fixed, $1 \leq t \leq \infty$.

The precursor to this problem is the classical *closest-pair problem* which is to compute the closest pair of points in a static set S, $|S| = n$. Shamos and Hoey [12] and Bentley and Shamos [2] gave $O(n \log n)$ time algorithms for solving the closest-pair problem in the plane and in arbitrary but fixed dimension, respectively. This running time is optimal in the algebraic decision tree model [1]. If we allow randomization as well as the use of the (non-algebraic) floor function, we find algorithms with better (expected) running times for the closest-pair problem. Rabin, in his seminal paper [9] on randomized algorithms, gave an $O(n)$ expected time algorithm for this problem. A different approach, leading to a simpler algorithm also with $O(n)$ expected running time, was recently described by Khuller and Matias [7]. A randomized "sieving" procedure described in this paper is at the heart of our dynamic algorithm.

There has been a lot of work on maintaining the closest pair of a dynamically changing set of points. When restricted to the case where only insertions of points are allowed (sometimes known as the *on-line closest-pair problem*) a series of papers culminated in an optimal data structure due to Schwarz, Smid and Snoeyink [11]. Their data structure required $O(n)$ space and supported insertions in $O(\log n)$ time.

The existing results are not as satisfactory when deletions must be performed. If only deletions are to be performed, Supowit [15] gave a data structure with $O(\log^k n)$ amortized update time that uses $O(n \log^{k-1} n)$ space. When both insertions and deletions are allowed, Smid [14] described a data structure that uses $O(n \log^k n)$ space and runs in $O(\log^k n \log\log n)$ amortized time per update. Another data structure due to Smid [13], with improvements stemming from results of Salowe [10] and Dickerson and Drysdale [5], uses $O(n)$ space and requires $O(\sqrt{n} \log n)$ time for updates; this is the best linear-space data structure currently known for insertions and deletions.

In this paper we discuss a randomized data structure, the *sparse partition*, which solves the dynamic closest pair problem in arbitrary fixed dimension using $O(\log n)$ expected time per update. The data structure needs $O(n)$ expected space. We assume that the updates are generated by an adversary who can insert or delete arbitrary points but has no knowledge of the random choices that the algorithm makes. The above bound is obtained assuming the use of the floor function and assuming that there is some prior bound on the size of the points (in order to make possible the use of hashing). If we want to dispense with hashing, then the algorithm can be modified to run in $O(\log n \log\log n)$ expected time per update. If we remove both assumptions, we obtain an algorithm with $O(\log^2 n)$ expected

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

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

time per update in the algebraic decision tree model [1]. All three versions of the randomized algorithm are more efficient than the currently best known deterministic algorithms for solving the problem. Indeed our algorithm is the first to obtain polylogarithmic update time using linear space for the dynamic closest-pair problem.

The sparse partition is a random structure; given a set S of points, the structure that stores S will be randomly chosen from a set of many possible structures. In one version of the data structure, the probability that a particular structure is the one that is being used will depend only on the set S that is being stored and not upon the sequence of insertions and deletions that were used to construct S. In this sense, the data structure is reminiscent of skip-lists or randomized search trees.

2 Sparse partitions

Let S be a set of n points in k-dimensional space. Let $1 \leq t \leq \infty$. We denote the L_t-distance between the points p and q by $d(p,q)$. The *minimal distance* of S is $\delta(S) := \min\{d(p,q) : p,q \in S, p \neq q\}$. A *closest pair* in S is a pair $p,q \in S$ such that $d(p,q) = \delta(S)$. The distance of p to its nearest neighbor in S is denoted by $d(p,S) := \min\{d(p,q) : q \in S \setminus \{p\}\}$.

In this section, we define the notion of a sparse partition. This definition is independent of the implementation. In later sections, we shall give two ways to implement such sparse partitions.

DEFINITION 2.1. A *sparse partition* for the set S is a sequence of 5-tuples $(S_i, S'_i, p_i, q_i, d_i)$, $1 \leq i \leq L$, where L is a positive integer, such that:

(a) For $i = 1, \ldots, L$:
  (a.1) $S_i \neq \emptyset$;
  (a.2) $S'_i \subseteq S_i \subseteq S$;
  (a.3) $p_i, q_i \in S_i$ and $p_i \neq q_i$ if $|S_i| > 1$;
  (a.4) $d_i = d(p_i, q_i) = d(p_i, S_i)$.

(b) For all $1 \leq i \leq L$, and for all $x \in S_i$:
  (b.1) If $d(x, S_i) > d_i/3$ then $x \in S'_i$;
  (b.2) If $d(x, S_i) \leq d_i/6k$ then $x \notin S'_i$.

(c) For all $1 \leq i < L$, and for all $x \in S_i$:
  If $x \in S_{i+1}$, then there is a point in $y \in S_i$ such that $d(x,y) \leq d_i/3$ and $y \in S_{i+1}$.

(d) $S_1 = S$ and for $1 \leq i \leq L-1$, $S_{i+1} = S_i \setminus S'_i$.

DEFINITION 2.2. A set S stored by a sparse partition is said to be *uniformly stored* if the sparse partition storing it has the property that for all $i = 1, \ldots, L$ and all $x \in S_i$, $\Pr(x = p_i) = 1/|S_i|$.

LEMMA 2.1. *Any sparse partition for S satisfies the following properties:*
(1) *The sets S'_i, for $1 \leq i \leq L$, are non-empty and pairwise disjoint. For any $1 \leq i \leq L$, $S_i = \bigcup_{j \geq i} S'_j$. In particular, $S'_1 \cup \ldots \cup S'_L$ is a partition of S.*
(2) *For any $1 \leq i < L$, $d_{i+1} \leq d_i/3$. Moreover, $d_L/6k \leq \delta(S) \leq d_L$.*

Proof. For (1), we only need to prove that $S'_i \neq \emptyset$ for all i. (The other claims are clear.) Since $p_i \in S_i$ and $d(p_i, S_i) = d_i > d_i/3$, it follows from Condition (b.1) in Definition 2.1 that $p_i \in S'_i$.

To prove the first part of (2), let $1 \leq i < L$. Since $p_{i+1} \in S_{i+1}$, we know from Condition (c) in Definition 2.1 that there is a point $y \in S_i$ such that $d(p_{i+1}, y) \leq d_i/3$ and $y \in S_{i+1}$. Therefore,

$$d_{i+1} = d(p_{i+1}, S_{i+1}) \leq d(p_{i+1}, y) \leq d_i/3.$$

To prove the second part of (2), let p, q be a closest pair in S. Let i and j be such that $p \in S'_i$ and $q \in S'_j$. Assume w.l.o.g. that $i \leq j$. Then it follows from (1) that p and q are both contained in S_i. It is clear that $\delta(S) = d(p,q) = d(p, S_i)$. Condition (b.2) in Definition 2.1 implies that $d(p, S_i) > d_i/6k$. Since the d_l's are decreasing, we conclude that $\delta(S) > d_i/6k \geq d_L/6k$. The inequality $\delta(S) \leq d_L$ obviously holds, because d_L is a distance between two points of S. ∎

We now give an algorithm that, given an input set S, stores it uniformly as a sparse partition:

Algorithm *Build*(S)**:**
(i) Set $S_1 = S$; $i = 1$.

(ii) Choose a random point $p_i \in S_i$. Calculate $d_i = d(p_i, S_i)$. Let $q_i \in S_i$ be such that $d(p_i, q_i) = d_i$.

(iii) Choose S'_i to satisfy (b.1), (b.2) and (c) in Definition 2.1.

(iv) If $S_i = S'_i$ stop; otherwise set $S_{i+1} = S_i \setminus S'_i$, set $i = i + 1$ and goto (ii).

LEMMA 2.2. *Let S be a set containing n points. Run Build(S) and let S_i, $1 \leq i \leq L$, be the sets constructed by the algorithm. Then* $\mathrm{E}(\sum_{i=1}^{L} |S_i|) \leq 2n$.

Proof. We first note that $L \leq n$ by Lemma 2.1. Define $S_{L+1} := S_{L+2} := \ldots := S_n := \emptyset$. Let $s_i := \mathrm{E}(|S_i|)$ for $1 \leq i \leq n$. We will show that $s_{i+1} \leq s_i/2$, which implies that $s_i \leq n/2^{i-1}$. By the linearity of expectation, we get $\mathrm{E}(\sum_{i=1}^{L} |S_i|) \leq \sum_{i=1}^{n} n/2^{i-1} \leq 2n$.

It remains to prove that $s_{i+1} \leq s_i/2$. If $s_i = 0$, then $s_{i+1} = 0$ and the claim holds. So assume $s_i > 0$. We consider the conditional expectation $\mathrm{E}(|S_{i+1}| \mid |S_i| = l)$. Let $r \in S_i$ such that $d(r, S_i) \geq d_i$. Then, Condition (b.1) of Definition 2.1 implies that $r \in S'_i$, i.e., $r \notin S_{i+1}$.

Take the points in S_i and label them $r_1, r_2, \ldots, r_l$ such that $d(r_1, S_i) \leq d(r_2, S_i) \leq \ldots \leq d(r_l, S_i)$. The point p_i is chosen randomly from the set S_i, so it can be any of the r_j's with equal probability. Thus $\mathrm{E}(|S_{i+1}| \mid |S_i| = l) \leq l/2$, from which it follows that $s_{i+1} = \sum_l \mathrm{E}(|S_{i+1}| \mid |S_i| = l) \cdot \Pr(|S_i| = l) \leq s_i/2$. ∎

DEFINITION 2.3. Let $S'_1, S'_2, \ldots, S'_L$ be the sets of a sparse partition for S. For any $p \in \mathbb{R}^k$ and $1 \leq i \leq L$, define the *restricted distance*

$$d_i^*(p) := \min\left(d_i,\ d(p, \cup_{j \leq i} S'_j)\right),$$

i.e., the smaller of d_i and the minimal distance between p and all points in $S'_1 \cup S'_2 \cup \ldots \cup S'_i$.

LEMMA 2.3. *Let $p \in S$ and let i be the index such that $p \in S'_i$.*
(1) $d_i^*(p) > d_i/6k$.
(2) *If $q \in S'_j$, where $j < i-k$, then $d(p,q) > d_i$.*
(3) $d_i^*(p) = \min\left(d_i,\ d(p, S'_{i-k} \cup S'_{i-k+1} \cup \ldots \cup S'_i)\right)$.

Proof. (1) Let $1 \leq j \leq i$ and let $q \in S'_j$. Since $p \in S_j$, it follows from Condition (b.2) of Definition 2.1 that $d(p,q) \geq d(q, S_j) > d_j/6k \geq d_i/6k$.

(2) Let $q \in S'_j$, where $j < i-k$. As in (1), we get $d(p,q) > d_j/6k$. Then, Lemma 2.1 implies that $d(p,q) > \frac{d_j}{6k} \geq \frac{d_{i-k-1}}{6k} \geq \frac{3^{k+1} d_i}{6k} > d_i$. Finally, (3) follows immediately from (2). ∎

LEMMA 2.4.

$$\delta(S) = \min_{1 \leq i \leq L} \min_{p \in S'_i} d_i^*(p) = \min_{L-k \leq i \leq L} \min_{p \in S'_i} d_i^*(p).$$

Proof. The value $d_i^*(p)$ is always the distance between two points in S. Therefore, $\delta(S) \leq \min_{1 \leq i \leq L} \min_{p \in S'_i} d_i^*(p)$. Let p, q be a closest pair with $p \in S'_i$ and $q \in S'_j$. Assume w.l.o.g. that $j \leq i$. Clearly, $d(p,q) = d(p, \cup_{h \leq i} S'_h) \geq d_i^*(p)$. This implies that $\delta(S) \geq \min_{1 \leq i \leq L} \min_{p \in S'_i} d_i^*(p)$, proving the first equality.

It remains to prove that we can restrict the value of i to $L-k, L-k+1, \ldots, L$. We know from Lemma 2.3 (1) that $\min_{p \in S'_i} d_i^*(p) > d_i/6k$. Moreover, we know from Lemma 2.1 (2), that for $i < L-k$, $d_i/6k \geq d_{L-k-1}/6k \geq (3^{k+1}/6k) \cdot d_L > d_L \geq \delta(S)$. ∎

In order to be able to find the minimal distance $\delta(S)$, we maintain the following information. For each $1 \leq i \leq L$, we compute the restricted distances $\{d_i^*(p) : p \in S'_i\}$. (How we compute these values depends on the way we implement the sparse partition.) These distances are stored in a heap H_i, with the minimum at the top.

We now claim that, given these heaps, we can find the closest pair in constant time. Indeed, Lemma 2.4 says that $\delta(S)$ can only be stored in the heaps $H_{L-k}, H_{L-k+1}, \ldots, H_L$. To find $\delta(S)$ it is therefore enough to take the minima of these $k+1$ heaps and then to take the minimum of these $k+1$ values.

Now we can give an abstract description of our data structure.

The closest-pair data structure:
1. A data structure storing the sparse partition.
2. The heaps $H_1, H_2, \ldots, H_L$.

The rest of the paper is organized as follows. We discuss different ways to implement the data structure. First, we describe a grid based implementation. Since this data structure is the most intuitive one, we describe the update algorithms for this structure. Then, we define the other variants of the data structure. Concerning implementation details and update algorithms, we then only mention what changes have to be made in comparison to the grid based implementation to establish the results.

3 A grid based implementation

To give a concrete implementation of a sparse partition, we only have to define the set S'_i, i.e. the subset of sparse points in S_i, for each i.

Let S be a set of n points in k-dimensional space. We assume that the points of S are chosen from a bounded universe, more precisely: we assume that the ratio of their maximal and minimal distance is bounded. We start with some definitions.

Let $d > 0$. We use $G(d)$ to denote the grid with mesh size d and a lattice point at $(0, 0, \ldots, 0)$. Hyperrectangles of the grid are called *boxes*. More precisely, a box has the form

$$[i_1 d : (i_1+1)d) \times [i_2 d : (i_2+1)d) \times \ldots \times [i_k d : (i_k+1)d),$$

for integers $i_1, \ldots, i_k$. We call $(i_1, \ldots, i_k)$ the *index* of the box. Note that with this definition of a box as the product of half-open intervals, every point in $\mathbb{R}^k$ is contained in exactly one grid box.

The *neighborhood* of a box b in the grid $G(d)$, denoted by $N(b)$, consists of b itself plus the collection of $3^k - 1$ boxes bordering on it.

Let q be any point in $\mathbb{R}^k$ and let b_q denote the box of $G(d)$ that contains q. The *neighborhood of q in $G(d)$*, denoted by $N_d(q)$, is defined as the neighborhood of b_q, i.e. $N_d(q) := N(b_q)$.

We number the 3^k boxes in the neighborhood of q as follows. The number of a box is a k-tuple over $\{-1, 0, 1\}$. The j-th component of the k-tuple is -1,0, or 1, depending on wether the j-th coordinate of the box's index is smaller than, equal to or greater than the corresponding coordinate of b_q's index. We call this k-tuple the *signature* of a box. We denote by $b_d^\sigma(q)$ the box with signature σ in $N_d(q)$.With this notation, $q \in b_d^{0,\ldots,0}$. We are now going to define the notion of *partial neighborhood* of a point q. For any signature σ, we denote by $N_d^\sigma(q)$ the part of q's neighborhood that is in the neighborhood of $b_d^\sigma(q)$. Note that $N_d^\sigma(q)$

contains all the boxes $b_d^{\sigma'}(q)$ of $N_d(q)$ whose signature σ' differs from σ by at most 1 for each coordinate — these are exactly the boxes bordering on $b_d^{\sigma}(q)$ including $b_d^{\sigma}(q)$ itself. Particularly, $N_d^{0,\ldots,0}(q) = N_d(q)$, i.e. the partial neighborhood with signature $0,\ldots,0$ is the whole neighborhood of q.

Now we consider the neighborhood of a point $q \in \mathbb{R}^k$ restricted to a set of points. The *neighborhood of q in $G(d)$ relative to S*, denoted by $N_d(q,S)$, is the set of points in $S \setminus \{q\}$ that are contained in $N_d(q)$, the (unrestricted) neighborhood of q. We say that q *is sparse in $G(d)$ relative to S* if $N_d(q,S) = \emptyset$, i.e. if, besides q, there are no points of S in $N_d(q)$. In cases that S and d are understood from the context we will simply say that p is *sparse*.

We now list some properties of the neighborhood relation in grids. These properties will imply that the above definition of sparseness using the notion of neighborhood actually satisfies the requirements of a sparse partition according to Definition 2.1.

(N.1) If $N_d(p,S) = \emptyset$, then $d(p,S) > d$.
(N.2) For any $x \in N_d(p,S)$, $d(p,x) \leq 2k \cdot d$
(N.3) Let $S, T \subseteq \mathbb{R}^k$, and let $x \in S$ and $y \in T$. Then $x \in N_d(y,S) \iff y \in N_d(x,T)$.

If we are given d and S, then we use perfect hashing (see [4,6]) to store the points of S: For each point, we take as a key the index of the box in $G(d)$ that contains it. We store the keys of the non-empty boxes in a hash table. With each box b, we store a list containing the points in $S \cap b$, in arbitrary order. We call this *storing S according to $G(d)$*. The box indices must be bounded to make possible the use of hashing. This is the case when the ratio of the maximal distance and the minimal distance between any two points in S is bounded.

If S is stored according to $G(d)$, then we can answer the question *"are any points from S in box b?"* in $O(1)$ worst case time. Moreover, if the answer is yes, we can report all points in $S \cap b$ in time proportional to their number. By checking all boxes in the neighborhood of an arbitrary point q, we check in $O(1)$ time if q is sparse in S. So, by doing this for each point in S we can, in linear time, find the set of sparse points in S.

We are now in a position to define our grid based data structure. Recall that the only remaining task is to define the sets S_i' precisely. We do this by using the notion of sparseness as defined above: For $i \geq 1$, let $S_i' := \{p \in S_i : p \text{ sparse in } G(d_i/6k) \text{ relative to } S_i\}$.

The grid based data structure:

1. S_i stored according to $G(d_i/6k)$, $1 \leq i \leq L$.
2. S_i' stored according to $G(d_i/6k)$, $1 \leq i \leq L$.
3. The heaps H_1, H_2, ..., H_L.

Since we only use grids $G(d_i/6k)$, we will use short forms for grid-dependent objects like boxes and neighborhoods: E.g., $N_i(p,T)$ stands for $N_{d_i/6k}(p,T)$, the neighborhood of p in $G(d_i/6k)$ relative to T. We will often refer to the data structures associated with index i as *level i*.

LEMMA 3.1. *Using the above definition of S_i', we get a sparse partition according to Definition 2.1.*

Proof. We only have to prove Conditions (b) and (c) of Definition 2.1. Let $1 \leq i \leq L$ and let $x \in S_i$. First assume that $x \notin S_i'$. Then, there is a point $q \in S_i$ that is in the neighborhood of x. By (N.1), $d(x,S_i) \leq d(x,q) \leq 2k \cdot d_i/6k = d_i/3$. This proves Condition (b.1). To prove (b.2), assume that $x \in S_i'$. Then, the neighborhood of x relative to S_i is empty. Hence, by (N.2), $d(x,S_i) > d_i/6k$.

To prove (c), let $1 \leq i < L$ and let $x \in S_{i+1} = S_i \setminus S_i'$. It follows that there is a point $y \in S_i$ such that $y \in N_i(x,S_i)$. By the symmetry property (N.3), this is equivalent to $x \in N_i(y,S_i)$ and therefore $y \in S_{i+1}$. From Condition (b.1), we also have $d(x,y) \leq d_i/3$. This proves that we indeed have a sparse partition. ∎

This is probably a good time to point out that the grid implementation of our algorithm *Build* given in Section 2 is essentially the randomized static closest-pair algorithm given in [7] with many bells and whistles attached. The algorithm there was only concerned with finding d_L, since with it, one can find the closest-pair in $O(n)$ time (see [7] for details). It did not have to save the information at all of the levels. Our algorithm, in order to be dynamic, must have access to it.

LEMMA 3.2. *Assume we are given the grid based data structure for S. Let $1 \leq i \leq L$ and let $p \in S_i'$. The value $d_i^*(p)$ can be computed in $O(1)$ time.*

Proof. We know from Lemma 2.3 (2) that if $d_i^*(p) = d(p,q)$ with $d(p,q) < d_i$ then q must be in one of $S_i', S_{i-1}', \ldots, S_{i-k}'$. Furthermore, there are only a constant number of boxes in the grids $G(d_j/6k)$, $i-k \leq j \leq i$, where a point q can possibly appear in: these are the grid boxes that are within $6k$ boxes of the box that p is located in. Finally, because of the sparseness of S_j', $i-k \leq j \leq i$, in their respective sets, there can be at most one point found in each grid box. Therefore, using the hash tables storing S_j', $i-k \leq j \leq i$, we can find all these points and compute $d_i^*(p)$ in constant time. ∎

LEMMA 3.3. *The grid based data structure can be built in $O(n)$ expected time and it has $O(n)$ expected size. Given this data structure, we can find the closest pair in S in $O(1)$ time.*

Proof. Consider the i-th iteration of algorithm *Build*(S). Step (ii) can be performed in $O(|S_i|)$ deterministic time by calculating the distance between

p_i and all other points in S_i. Steps (iii)–(vi) take $O(|S_i| + |S_i'|) = O(|S_i|)$ expected time, because we use perfect hashing. Therefore the expected running time of the algorithm is bounded by $O(\mathrm{E}(\sum_i(|S_i|))$, which is also the amount of space used. Lemma 2.2 shows that this quantity is $O(n)$. Finally, we can compute the closest pair in S in constant time from the heaps, using Lemma 2.4 and Lemma 3.2. ∎

The grid-based data structure also has properties that extend Definition 2.1. These properties will be used for the dynamic mainenance of the data structure. The most important one is the following:

> For any $p \in \mathbb{R}^k$ and any $1 \le i < L$, if p is sparse in $G(d_i/6k)$ relative to S_i, then p is also sparse in $G(d_{i+1}/6k)$ relative to S_{i+1}.

This statement is equivalent to $N_i(x, S_i) = \emptyset \implies N_{i+1}(x, S_{i+1}) = \emptyset$, which will be shown in Lemma 3.5.

To establish the additional properties, we first examine the relationship between neighborhoods of different grids:

LEMMA 3.4. *Let $0 < d' \le d''/2$ be real numbers and let $q \in \mathbb{R}^k$. Then*
(N.4) $N_{d'}(q) \subseteq N_{d''}(q)$.
(N.5) *For any $\sigma \in \{-1, 0, 1\}^k$: $b^\sigma_{d'}(q) \subseteq N^\sigma_{d''}(q)$.*

LEMMA 3.5. *For any $1 \le i \le L$ and any $x \in \mathbb{R}^k$: $N_{i+1}(x, S_{i+1}) \subseteq N_i(x, S_i)$.*

Proof. Let $x \in \mathbb{R}^k$. Apply (N.4) from Lemma 3.4 with $d'' = d_i/6k, d' = d_{i+1}/6k$, noting that $S_{i+1} \subseteq S_i$, to obtain $N_{i+1}(x, S_{i+1}) \subseteq N_i(x, S_i)$. ∎

4 Dynamic maintenance of the grid based data structure

We first give an intuitive description of the insertion algorithm. Let S be the current set of points, and assume we want to insert the point q. Assume that S is uniformly stored in the sparse partition. We want to store $S \cup \{q\}$ uniformly in a sparse partition. By assumption, p_1 is a random element of $S_1 = S$. (We call p_1 the *pivot*.) Now, to generate a pivot for $S_1 \cup \{q\}$ it suffices to retain p_1 as pivot with probability $|S_1|/(|S_1| + 1)$ and to choose q instead with probability $1/(|S_1| + 1)$. If q is chosen, then we discard everything and run *Build*$(S_1 \cup \{q\})$, terminating the procedure. This happens, however, only with probability $1/(|S_1| + 1)$ and so the expected cost is $O(1)$.

Assume now that p_1 remains unchanged as the pivot. We now check to see if q_1 and, hence, d_1 have to be changed. First note that q can be the nearest neighbor of at most $3^k - 1 \le 3^k$ points in S_1. (See [3].) This means that d_1 can change only if p_1 is one of these points. Since we assumed that the adversary cannot see the coin tosses of the algorithm, and since p_1 is chosen uniformly from S_1, it follows that the probability of d_1 changing is at most $3^k/|S_1|$. If d_1 changes, we run *Build*$(S_1 \cup \{q\})$ and terminate the procedure. The expected cost of this is $O(1)$. The previous two steps are called "check for rebuild" in the later part of this section.

Assume now that p_1, q_1 and d_1 remain unchanged. Let us denote $S \cup \{q\}$ by $\widetilde{S}$. We now need to determine the set $\widetilde{S}_2$, which contains the non-sparse points in $\widetilde{S}_1 = \widetilde{S}$. If q is sparse in S_1, it will go into $\widetilde{S}_1'$, and nothing further needs to be done, that is, the tuples $(S_i, S_i', p_i, q_i, d_i)$, $2 \le i \le L$, remain the same. So, in this case, we can terminate the procedure. Otherwise, $\widetilde{S}_2$ contains q and possibly some points from S_1'. The set of points which are deleted from S_1' due to the insertion of q is called $down_1$. This completes the construction of the first 5-tuple.

Now we need to insert q and $down_1$ into S_2. Before we describe the algorithm any further, we should take a closer look at the *down* sets.

We define $down_0 := \emptyset$. Now assume that the insertion algorithm attempts to construct the 5-tuple for $\widetilde{S}_i$ without having made a rebuilding yet. Then, for $1 \le j \le i$, $down_j$ is defined as the set of points in S that are sparse in $G(d_j/6k)$ relative to S_j but that are not sparse in this grid relative to $\widetilde{S}_j$, i.e.

$$down_j = \{x \in S \setminus S_{j+1} : x \in \widetilde{S}_{j+1}\}. \tag{4.1}$$

The following lemma gives two equivalent characterizations for the sets $down_j, 1 \le j \le i$:

LEMMA 4.1. *Let $D_j := S_j' \cup down_{j-1}$. Then*

$$down_j = \{x \in D_j : x \in N_j(q, D_j)\}. \tag{4.2}$$

The set $D_j = S_j' \cup down_{j-1}$ is called the "candidate set" for $down_j$. We also have:

$$x \in down_j \iff N_j(x, \widetilde{S}) = \{q\}. \tag{4.3}$$

Proof. We omit the proof due to lack of space. The proof uses Lemma 3.1, saying that we have a sparse partition, and Lemma 3.5, which says that a point which is sparse at level i is also sparse at level $i + 1$. ∎

LEMMA 4.2. *Let the sets $down_0, \ldots, down_i$ be defined as described above. Then $|\bigcup_{1 \le j \le i} down_j| \le 3^k$.*

Proof. Assume that $x \in down_j$ for some $j \le i$. Then $x \in N_j(q)$ and $N_j(x, S) = \emptyset$ by Equations (1)-(3). Moreover, let $x \in b_j^\sigma(q)$. The partial neighborhood $N_j^\sigma(q)$ is the intersection of q's neighborhood with the neighborhood of x. Since $N_j(x, S) = \emptyset$, $N_j^\sigma(q)$ contains no point of $S \setminus \{x\}$. Now, let $y \in b_l^\sigma(q)$ for any $l > j$. Since $d_l \le d_{j+1} \le d_j/3$ by Lemma 2.1 (2), Lemma 3.4

gives $y \in N_j^\sigma(q)$. This means that at levels $j+1 \le l \le i$, there cannot be any point in $down_l$ with signature σ except x itself. (Note that a point can be in several *down* sets.) It follows that for each $\sigma \in \{-1,0,1\}^k$, the set of points x in S such that there exists a $j \in \{1,\ldots,i\}$ satisfying $x \in down_j$ as well as $x \in b_j^\sigma(q)$ contains at most one element. ∎

In particular, each single *down* set has constant size. Let us now continue with the construction of the 5-tuple for $\widetilde{S}_i$. We now are in a position to clear a small lie we told earlier; constructing $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{d}_i)$ from $(S_i, S'_i, p_i, q_i, d_i)$ requires that, instead of one, up to $3^k + 1$ points (q as well the points in $down_{i-1}$) be considered as new pivots, and also increases the chance of one of these points being closer to the old pivot than the pivot's previous closest neighbor, but this only increases the probabilities by a constant factor.

If no rebuilding takes place, we determine $\widetilde{S}'_i$, which is the set of sparse points in $S'_i \cup down_{i-1} \cup \{q\}$ in the grid $G(d_i/6k)$. Note that $S'_i \cup down_{i-1}$ is a sparse set in $G(d_i/6k)$ by Lemma 3.1 and Lemma 3.5. Therefore q is the only point that can cause a point of this set to become non-sparse. From Lemma 4.1, recall that $D_i := S'_i \cup down_{i-1}$ is the candidate set for $down_i$, and $down_i$ is the subset of D_i that is *not* part of $\widetilde{S}_i$.

We now have constructed $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{d}_i)$ and also determined $\widetilde{S}'_{i+1} = \widetilde{S}'_i \setminus \widetilde{S}_{i+1}$. If $\widetilde{S}'_{i+1} = \emptyset$, then we are finished with the insertion algorithm. Otherwise, we continue with the next level.

We now give an outline of the probabilistic analysis. We show that the expected run-time of an insert operation is $O(\log n)$, provided that the time to construct a 5-tuple without rebuilding is $O(1)$.

The expectation is taken both over the new coin tosses and over the expected state of the old data structure (this means in particular that the run-times of consecutive inserts are not independent).

Let the initial set of tuples be $(S_i, S'_i, p_i, q_i, d_i)$, $1 \le i \le n$, padding the sequence out with empty tuples if necessary. Let T_i be the time to construct $\widetilde{S}_i$ from S_i assuming no rebuilding has taken place while constructing $\widetilde{S}_1, \ldots, \widetilde{S}_{i-1}$. Clearly, the expected run-time X satisfies $X \le \sum_{i=1}^n T_i$. Let $N = \lceil \log n \rceil$. For $1 \le i \le N$, it holds that $T_i = O(|S_i|)$ expected time with probability $O(1/|S_i|)$ and $T_i = O(1)$ otherwise, and so $\mathrm{E}(T_i) = O(1)$, independently of the previous state of the data structure. It is fairly easy to see that $\sum_{i=N+1}^n T_i$ is bounded by $c \cdot |S_{N+1}|$ for some constant c. This means that:

$$\mathrm{E}(X) \le \sum_{i=1}^{N} \mathrm{E}(T_i) + \mathrm{E}(c \cdot |S_{N+1}|) = O(\log n)$$

since $\mathrm{E}(|S_{N+1}|)$ is $O(1)$.

We now give a more detailed description of the algorithm; this will show the above algorithm outline can indeed be efficiently implemented. As already mentioned, we denote by *level* i the data structures associated with index i.

Let us examine the point movements between the different levels during an insertion more closely. Assume that we are working at level i, where $i \ge 1$, i.e. we are constructing the 5-tuple for $\widetilde{S}_i$. The definition of the *down* sets implies that, at each level i, $down_{i-1}$ is the set of points in S that move at least down to level i, while $down_i$ is the set of points in S that move at least down to level $i+1$, where $down_0 := \emptyset$. More specifically,

(i) $x \in down_i \setminus down_{i-1}$ means x *starts moving* at level i, i.e. $x \in S'_i$ and $x \notin \widetilde{S}'_i$,
(ii) $x \in down_{i-1} \setminus down_i$ means x *stops moving* at level i, i.e. $x \notin S'_i$ and $x \in \widetilde{S}'_i$,
(iii) $x \in down_{i-1} \cap down_i$ means that x *moves through* level i, i.e. $x \notin S'_i$ and $x \notin \widetilde{S}'_i$.

For all the points satisfying (i) or (ii), we have to update all the heaps where the considered points disappear (i) or enter (ii). This task will be performed by the procedure change_heap, to be described later. Of course, the changes from the 5-tuple $(S_i, S'_i, p_i, q_i, d_i)$ to the 5-tuple $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{d}_i)$ also have to be performed in the data structures that actually store the 5-tuple. We omit these details and only show how to compute the set $down_i$ in constant time.

We already saw that the total complexity of the *down* sets is constant. In particular, each single *down* set has constant size. Now we show that, given the candidate set $D_i = S'_i \cup down_{i-1}$, where S'_i is stored according to grid $G(d_i/6k)$, we can compute $down_i$ in constant time. We use Equation (4.2) for this purpose, i.e. we show that $X_i = \{x \in S'_i \cup down_{i-1} : x \in N_i(q, S'_i \cup down_{i-1})\}$ can be constructed in constant time. We want to know for all $x \in S'_i \cup down_{i-1}$ wether $x \in N_i(q, S'_i \cup down_{i-1})$. Using symmetry this is equivalent to $q \in N_i(x, S'_i \cup down_{i-1} \cup \{q\})$. How do we perform the membership tests? The elements in S'_i are already stored at that level, whereas the elements in $down_{i-1} \cup \{q\}$ are not. We tentatively insert these points into the data structure storing the sparse set S'_i. This proves that we can find X_i in constant time.

The complete algorithm is given below. We maintain the invariant that, if we have constructed the 5-tuples $(\widetilde{S}_j, \widetilde{S}'_j, \widetilde{p}_j, \widetilde{q}_j, \widetilde{d}_j)$ for $1 \le j \le i$ without rebuilding, then $\widetilde{S}_i = S_i \cup down_{i-1} \cup \{q\}$. Note that the invariant is true in the beginning because $down_0 = \emptyset$. The invariant ensures that the new set of 5-tuples satisfies the requirements of a sparse partition of Definition 2.1

as well as the properties of the neighborhood relation (N.1)-(N.5).

Algorithm Insert(q):
1. initialize $i \leftarrow 1$; $down_0 \leftarrow \emptyset$;

Invariant: $\widetilde{S}_i = S_i \cup down_{i-1} \cup \{q\}$

From the invariant, we know $\widetilde{S}_i$; we want to determine $\widetilde{S}'_i$. Before that, we check if the data structure has to be rebuilt, in which case the algorithm runs Build($\widetilde{S}_i$) and stops.

2. check for rebuild: flip an $\widetilde{S}_i$-sided coin, giving pivot $\widetilde{p}_i$ of $\widetilde{S}_i$; if $\widetilde{p}_i \notin S_i$, then Build($\widetilde{S}_i$); stop;

otherwise, the old pivot p_i of S_i is also the pivot for $\widetilde{S}_i$;

if $d(p_i, x) < d_i$ for some $x \in down_{i-1} \cup \{q\}$ then Build($\widetilde{S}_i$); stop;

otherwise, $d_i = d(p_i, S_i)$ equals $d(p_i, \widetilde{S}_i)$.

3. Determine $\widetilde{S}'_i$: To determine $\widetilde{S}'_i$ and therefore $\widetilde{S}_{i+1}$, it basically suffices to determine $down_i$, the set of points that started moving before level i and that move below level i.

We can compute $down_i$ in constant time as described above. The portion of $S'_i \cup down_{i-1}$ that is not in $down_i$ is sparse in $\widetilde{S}_i$ and will therefore go into $\widetilde{S}'_i$. So, out of $\widetilde{S}_i = S_{i+1} \cup S'_i \cup down_{i-1} \cup \{q\}$, we have determined all the elements w.r.t. membership in $\widetilde{S}'_i$, except q. If $down_i \neq \emptyset$, q is certainly not in $\widetilde{S}'_i$. Otherwise, we have to check $N_i(q, S_i)$, the neighborhood of q relative to S_i, separately to see if there are points in S_{i+1} that prevent q from being sparse in $\widetilde{S}_i$. In either case, we know $\widetilde{S}_{i+1}$. If q is not sparse in $\widetilde{S}_i$, then we have $\widetilde{S}_{i+1} = \widetilde{S}_i \setminus \widetilde{S}'_i = S_{i+1} \cup (S'_i \cup down_{i-1}) \setminus \widetilde{S}'_i \cup \{q\} = S_{i+1} \cup down_i \cup \{q\}$, i.e., the invariant is still valid for the next level.

4. Update heaps: Some points of $down_{i-1}$ may not be in $down_i$, and vice versa. As described before, these are the points that stop and start moving down, respectively. For these points we have to update the heaps. Let p be one of these points. We execute procedure change_heap(p), which is given below. We also insert q into the heap structure, if appropriate, using change_heap(q). Note that, at each level i, a point can be associated with only a constant number of heap values, which are located in the heaps $H_{i+l}, 0 \leq l \leq k$.

5. Next iteration: We are done with level i.
If $N_i(q, \widetilde{S}_i) \neq \emptyset$, then $i \leftarrow i + 1$; goto **2.**

Otherwise, stop. We have computed the sparse partition for the new set $\widetilde{S} = S \cup \{q\}$, and have also updated the heaps.

It remains to decribe the procedure change_heap(p) which actually performs the heap updates. There are two cases: the procedure is called (i) when p starts moving at level i and (ii) when p stops moving at some level j, where $i < j$. In the first case, we basically perform a deletion of the heap values associated with p, while in the second case, we perform the corresponding reinsertions into the heap structure. Note that the latter case does not occur if the data structure has been rebuilt at some level $i < l \leq j$. In this case, the rebuilding algorihm inserts the values associated with p into the heap structure appropriately.

From Lemma 2.3 (3), we know that $d_i^*(p) = \min\left(d_i, d(p, S'_{i-k} \cup \ldots \cup S'_i)\right)$. Thus, a point $p \in S'_i$ can be associated to a heap in two different ways: First, there is a value $d_i^*(p)$ in H_i. Furthermore, for each $0 \leq l \leq k$, there may be $q \in S'_i + l$ such that $d(q, p)$ gives rise to $d_i^*(q) \in H_{i+l}$. The procedure is as follows:

```
if p starts moving at level i, (p ∈ S'_i, but p ∉ S̃'_i) then
  delete d*_i(p) from H_i;
  for 0 ≤ l ≤ k do:
    for all q ∈ S'_{i+l} such that d*_{i+l}(q) = d(q,p) do:
      recompute d*_{i+l}(q); replace old value;
else (p stops moving at level j, i.e. p ∉ S'_j, but p ∈ S̃'_j)
  compute d*_j(p); insert it into H_j;
  for 0 ≤ l ≤ k do:
    for all q ∈ S̃'_{j+l} such that d(q,p) < d*_{j+l}(q) do:
      d*_{j+l}(q) = d(q,p); replace old value;
```

We haven't mentioned yet how the heaps are connected with the points stored in the sparse partition. For each value $d_i^*(p)$ in H_i, we store a pointer from the unique occurrence of p in S'_i to it. Moreover, with $d_i^*(p)$ we store the pair (p, q) such that $d_i^*(p) = d(p, q)$. These informations make accessible the heap values that are involved in the procedure.

Each restricted distance can be computed in $O(1)$ time by Lemma 3.2. Moreover, from the proof of Lemma 3.2 we know that the restricted distances are computed by searching the area of at most $6k$ boxes away from p in the grids that store the sparse sets $S'_{i+l}, 0 \leq l \leq k$. Outside this area, the restricted distance of a point q cannot be affected by removal of p. Since we assume that the dimension k is fixed, the total number of heap operations carried out by the procedure is constant.

THEOREM 4.1. *Insert(q) correctly maintains the data structure and takes expected time $O(\log n)$.*

Proof. As discussed before, the algorithm maintains a sparse partition according to Definition 2.1, and the expected cost of step 2, summed over the entire procedure, is $O(\log n)$.

The running time for step 3 is $O(1 + |down_{i-1}| +$

$|down_i|$) at level i. Since $|down_i| = O(1)$, $1 \le i \le L$, each step costs only constant time and the cost of this step is subsumed in the cost of step 2.

Now consider step 4. From Lemma 4.2 we know that the heap update procedure is only called for a constant number of points over all the levels. Since this procedure only performs $O(1)$ heap operations, the total time for the heap updates is $O(\log n)$: $O(1)$ heap operations, each of cost $O(\log n)$. ∎

Now we come to the algorithm that deletes a point q from the data structure. Let $\widetilde{S}$ denote $S \setminus \{q\}$ during the discussion of deletion. Deletion is basically the reverse of insertion. In particular, the points that move to lower levels during an insertion of q move back to their previous locations when q is deleted directly afterwards, if no rebuilding takes place. Because of the rebuilding, we have to be a little more careful with deletions. An insertion ends at the level where the new point q eventually is sparse. Therefore, we want to start the deletion at the level h such that $q \in S'_h$. Since q has to be deleted from all the sets S_i, $1 \le i \le h$, we have to take care that, by doing this, we don't delete the pivot or its nearest neighbor at some of these levels. For this purpose, we first attempt to find the level h such that $q \in S'_h$, starting at the first level. If the pivot conditions mentioned above are violated at some level $i \le h$, we rebuild at that level and stop without having located q in the set S'_h. Then we walk up again and delete q from the levels encountered. Note that no rebuilding occurs any more in this phase.

In order to be able to delete q efficiently from the non-sparse sets S_i containing it, we link the occurrence of a point in S_i to its occurrence in S_{i-1} and vice versa, if the corresponding level exists.

As already mentioned, points may move up some levels due to the deletion of q. Let $1 \le i \le h$ and $q \in S'_h$. Then $q \in S_i$. Analogously to the insertion algorithm, we define up_i to be the set of points the movement of which started below level i and does not stop before level i, i.e. $up_i = \{x \in S_{i+1} \setminus \{q\} : x \notin \widetilde{S}_{i+1}\}$, if no rebuilding takes place at a level $j \le i$. Otherwise, $up_i := \emptyset$. Concerning the number of points moving between levels during an update operation, the *up* sets are identical to the *down* sets, i.e. $|\bigcup_{1 \le i \le L} up_i| \le 3^k$, see Lemma 4.2.

We can compute a set up_i in constant time, as follows. The corresponding statement to Equation (4.3) is $x \in up_i \iff N_i(x, S_i) = \{q\}$. Checking this condition means finding all points in S_i having only q in their neighborhood. Using the symmetry property (N.3), this can be done in $O(1)$ time.

Analogous to the insertion algorithm, (i) $x \in up_{i-1} \setminus up_i$ means x *starts moving* at level i, i.e. $x \in S'_i$ and $x \notin \widetilde{S}'_i$, (ii) $x \in up_i \setminus up_{i-1}$ means x *stops moving* at level i, i.e. $x \notin S'_i$ and $x \in \widetilde{S}'_i$, and (iii) $x \in up_{i-1} \cap up_i$ means that x *moves through* level i, i.e. $x \notin S'_i$ and $x \notin \widetilde{S}'_i$. As before, the points that start or stop moving are causing heap updates.

Algorithm Delete(q):

1. check for rebuild: $i \leftarrow 1$;

while $q \notin S'_i$ and no rebuilding takes place, do:

if q is the pivot p_i or if $d(p_i, q) < d_i$
then Build($S_i \setminus \{q\}$); else $i \leftarrow i + 1$;

Now, either $q \in S'_i$, or the data structure has been rebuilt for $S_i \setminus \{q\}$ and q was previously stored in $S'_h, h \ge i$. In either case, we have $up_i = \emptyset$.

We know up_i and $\widetilde{S}_{i+1}$, we want to determine $\widetilde{S}_i$.

2. Determine $\widetilde{S}_i$: We are interested in up_{i-1}, i.e. the set of points in $S_i \setminus \{q\}$ that don't go into $\widetilde{S}_i$. From the discussion above, we can compute up_{i-1} in constant time using $up_{i-1} = \{x \in S_{i-1} : N_{i-1}(x, S_{i-1}) = \{q\}\}$. Now we know $\widetilde{S}_i = S_i \setminus up_{i-1} \setminus \{q\}$ and $\widetilde{S}'_i = \widetilde{S}_i \setminus \widetilde{S}_{i+1}$. In particular, we know that all $x \in up_i \setminus up_{i-1}$ go into $\widetilde{S}_i$ and $\widetilde{S}'_i$.

4. Update heaps: Completely analogous to Algorithm Insert. We execute change_heap for the points in the symmetric difference of up_{i-1} and up_i, and for the deleted point q, if we are on a level where q contibutes a heap value.

5. Next iteration: If $i > 1$, then $i \leftarrow i - 1$; goto **2**.

Otherwise, stop. We have computed the sparse partition for the new set $\widetilde{S} = S \setminus \{q\}$ and updated the heaps.

THEOREM 4.2. *Delete(q) correctly maintains the data structure and takes expected time $O(\log n)$.*

Proof. The proofs of correctness and running time are analogous to those for the insertion algorithm and are therefore omitted. ∎

5 Removing the finite universe assumption

In the previous sections, we stored the non-empty grid boxes using perfect hashing. Therefore, we required that the indices of the boxes are from a finite universe. Clearly, we can also store the non-empty boxes in a balanced binary search tree. Given a point p, we use the floor function to find the box that contains this point. Then, we search for this box in logarithmic time. Similarly, we can insert and delete points: If a new point is contained in a new box, we insert the box, together with the point; otherwise, we add the point to the box that is stored in the tree already.

For the update algorithms, we need an expected number of $O(\log n)$ dictionary operations plus $O(\log n)$

time per update operation. Now, each dictionary operation takes $O(\log n)$ time. Hence, the expected update time is increased to $O(\log^2 n)$. Clearly, for this solution, we do not need the finiteness of the universe; it works for arbitrary point sets.

Note that during an update, we perform basically the same search operations at each level. Therefore, we can apply a special form of fractional cascading to improve the expected update time. This leads to the following result. (Details will be given in the full paper.)

THEOREM 5.1. *For the dynamic closest pair problem, with arbitrary point sets, there exists a randomized data structure of expected size $O(n)$, that maintains the closest pair in $O(\log n \log\log n)$ expected time per insertion and deletion. The algorithms on this data structure use the floor function.*

6 An algebraic decision tree implementation

The solution of the previous section still uses the floor function. It is well known that this 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 $O(n)$ complexity.

Therefore, we want an algorithm that does not use the floor function. Note that this function was only used to compute the grid box containing a given point. Therefore, we will use a degraded grid for which we only need algebraic functions. A similar type of grid appears already in [8].

DEFINITION 6.1. Let S be a set of n points in k-dimensional space and let d be a positive real number. A k-dimensional degraded d-grid is a collection of hyperrectangles, that are inductively defined as follows:
(1) For $k = 1$, a one-dimensional degraded d-grid is a set of intervals $s_i = [a_i, b_i)$, $i \geq 1$, that are non-empty, pairwise disjoint and cover S, such that $d \leq b_i - a_i \leq 2d$ and $a_{i+1} - b_i \geq d$.
(2) Let $k > 1$. Let $s_1, s_2, \ldots$ be a one-dimensional degraded d-grid for the multiset consisting of the first coordinates of all points in S. Let $S^{(i)}$ be the points of S whose first coordinates are contained in s_i. Let $b_{i1}, b_{i2}, \ldots$ be the hyperrectangles of a $(k-1)$-dimensional degraded d-grid for $S^{(i)}$, where we only take the last $k-1$ coordinates. Then the collection of hyperrectangles $s_i \times b_{ij}$, $i, j \geq 1$, is a k-dimensional degraded d-grid for S.

In [8], an algorithm is given that constructs a k-dimensional degraded d-grid in $O(n \log n)$ time. It follows from Definition 6.1, that we can store a degraded d-grid using the slab method. In particular, we can locate the hyperrectangle containing a point p, by performing k binary searches. Hence, it takes $O(\log n)$ time to locate a point.

Suppose we want to insert a new point $p = (p_1, \ldots, p_k)$. Then, we first check if there is already an interval s_i containing p_1. If so, we insert $(p_2, \ldots, p_k)$ into the $(k-1)$-dimensional degraded d-grid for $S^{(i)}$, using the same algorithm recursively.

Otherwise, there is no interval containing p_1. The search for p ended between two non-empty intervals, say s and t. If the interval between s and t has width less than $2d$, we make one new (non-empty) interval out of it. We insert this new interval into the data structure. Otherwise, the interval between s and t has width at least $2d$. We make a new interval, say u, taking care that the interval between s and u has width either zero, or at least d. Similarly, the interval between u and t has width either zero or at least d. As before, we insert u into the data structure. The entire insert algorithm takes $O(\log n)$ time. The delete algorithm, which also takes $O(\log n)$ time, is similar.

In order to implement our data structure, we define the sets S_i'. First, we need some definitions. Let $g_i := d_i/42k$. We store the set S_i in a degraded g_i-grid. Each rectangle in this degraded grid has sides of length between g_i and $2g_i$. Note that the degraded grid not only depends on g_i, as in the grid case, but also on the set S_i to be stored in it. (Actually, it even depends on the way S_i has developed by updates.) Let B be a rectangle of this degraded grid and let $(b_1, b_2, \ldots, b_k)$ be the "lower left" corner of B. Then $N(B)$, the neighborhood of B, is defined as the set of all rectangles of the degraded grid that have their lower left corner in the box

$$[b_1 - 9g_i : b_1 + 11g_i] \times \ldots \times [b_k - 9g_i : b_k + 11g_i].$$

The other definitions are analogous to the ones given in Section 3. Let p be a point in $\mathbb{R}^k$ and let B_p be the rectangle of the degraded grid that contains p. Then the neighborhood of p in the degraded grid relative to S_i, denoted by $N_i(p, S_i)$, is defined as the set of all points in $S_i \setminus \{p\}$ that are contained in any of the rectangles in $N(B_p)$, the neighborhood of B_p. As before, we say that a point p is sparse in the degraded grid relative to S_i if $N_i(p, S_i) = \emptyset$. We define $S_i' := \{p \in S_i : p \text{ sparse in the degraded } g_i\text{-grid relative to } S_i\}$.

The degraded grid based data structure:

1. S_i stored in a degraded g_i-grid, $1 \leq i \leq L$;
2. S_i' stored in a degraded g_i-grid, $1 \leq i \leq L$;
3. The heaps $H_1, H_2, \ldots, H_L$.

LEMMA 6.1. *Using this definition for S_i', we get a sparse partition according to Definition 2.1.*

Proof. Let $1 \leq i \leq L$ and let $x \in S_i$. Assume that $x \in S'_i$. Then $d(x, S_i) > 9g_i - 2g_i = 7g_i = d_i/6k$. If $x \notin S'_i$, then $d(x, S_i) \leq 11g_ik + 2g_ik = 13g_ik \leq d_i/3$. This proves that Condition (b) of Definition 2.1 holds. It remains to prove Condition (c). Let $1 \leq i < L$ and let $x \in S_{i+1} = S_i \setminus S'_i$. Let y be the nearest neighbor of x in S_i. Since the neighborhood relation for rectangles of the degraded grid is symmetric, the neighborhood relation for points is also symmetric. Therefore, y is not sparse relative to S_i, i.e., $y \in S_{i+1}$. ∎

Note that Lemma 6.1 corresponds to Lemma 3.1 in Section 3. We also have an analogous statement to Lemma 3.5:

LEMMA 6.2. *For any $1 \leq i \leq L$ and any $x \in \mathbb{R}^k$: $N_{i+1}(x, S_{i+1}) \subseteq N_i(x, S_i)$.*

Proof. Let B_i and B_{i+1} denote the box containing x in the degraded g_i-grid for S_i and in the degraded g_{i+1}-grid for S_{i+1}, respectively. We want to show that $N(B_{i+1}) \subseteq N(B_i)$, from which the lemma follows.

Surely, $P_i := [p_1 - 9g_i : p_1 + 9g_i] \times \ldots \times [p_k - 9g_i : p_k + 9g_i] \subseteq N(B_i)$. Similarly, $P_{i+1} := [p_1 - 11g_{i+1} : p_1 + 11g_{i+1}] \times \ldots \times [p_k - 11g_{i+1} : p_k + 11g_{i+1}] \supseteq N(B_{i+1})$. But $g_{i+1} \leq g_i/3$ from Lemma 6.1, and so $P_{i+1} \subseteq P_i$, which implies $N(B_{i+1}) \subseteq N(B_i)$. ∎

Now let us examine the update algorithms. In Section 4, we defined the sets $down_i$. The definition remains the same here, except that we have a new notion of neighborhood. Note that the 3 equivalent characterizations of the *down* sets in Equations (4.1)-(4.3) are still valid, since they only relied on Lemmas 3.1 and 3.5, for which we have proved the corresponding statements in Lemmas 6.1 and 6.2.

This means that the correctness proofs are still valid. Concerning the complexity of the *down* sets, we only prove a very crude bound:

LEMMA 6.3. *Let $1 \leq i \leq L$ and $down_j$ be defined for $1 \leq j \leq i$. Then $|down_j| \leq 20^k$, for each j.*

Proof. Let B_q denote the box containing q in the degraded g_j-grid for S_j. From Equations (4.1)-(4.3), we infer that each $x \in down_j$ must satisfy $x \in N(B_q)$ and $N_j(p, S_j) = \emptyset$. Particularly, each of these points must be the only one in its box. So, each of the at most 20^k boxes of $N(B_q)$ can contain at most one point of S. ∎

It follows that we have logarithmic bound on the expected size of the union of the *down* sets. Hence, there are $O(\log n)$ heap operations per update. These already take $O(\log^2 n)$ time. The same holds for the *up* sets that come up during the deletion algorithm.

All the other lemmas and theorems concerning correctness and running times remain valid, except that we have to multiply the time bounds by $\log n$.

We remark here that this data structure does no longer have the property that its distribution is independent of the history of updates. The analysis, however, remains essentially unchanged.

THEOREM 6.1. *Let S be a set of n points in $\mathbb{R}^k$. There exists a randomized data structure of expected size $O(n)$, that maintains the closest pair in S in $O(\log^2 n)$ expected time per insertion and deletion. The algorithms on this data structure fit in the algebraic decision tree model.*

References

[1] M. Ben-Or. *Lower bounds for algebraic decision trees.* Proc. 15th STOC, 1983, pp. 80-86.

[2] J.L. Bentley and M.I. Shamos. *Divide-and-conquer in multidimensional space.* Proc. 8th STOC, 1976, pp. 220-230.

[3] W.H.E. Day and H. Edelsbrunner. *Efficient algorithms for agglomerative hierarchical clustering methods.* Journal of Classification **1** (1984), pp. 7-24.

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

[5] M.T. Dickerson and R.S. Drysdale. *Enumerating k distances for n points in the plane.* Proc. 7th ACM Symp. Comp. Geom., 1991, pp. 234-238.

[6] M. Fredman, F. Komlos and E. Szemeredi. *Storing a sparse table with O(1) worst case access time.* Journal of the ACM **17** (1984), pp. 538-544.

[7] S. Khuller and Y. Matias, *A simple randomized sieve algorithm for the closest-pair problem.* Proc. 3rd Can. Conf. Comp. Geom., 1991, pp. 130-134.

[8] H.P. Lenhof and M. Smid. *Enumerating the k closest pairs optimally.* Proc. 33rd FOCS, 1992.

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

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

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

[12] M.I. Shamos and D. Hoey. *Closest-point problems.* Proc. 16th FOCS, 1975, pp. 151-162.

[13] M. Smid. *Maintaining the minimal distance of a point set in less than linear time.* Algorithms Review **2** (1991), pp. 33-44.

[14] ———. *Maintaining the minimal distance of a point set in polylogarithmic time.* Discrete Comput. Geom. **7** (1992), pp. 415-431.

[15] K.J. Supowit. *New techniques for some dynamic closest-point and farthest-point problems.* Proc. 1st SODA, 1990, pp. 84-90.