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 D-dimensional space, for any fixed D, can be found in constant time. If a frame containing all the points is known in advance, 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. This method is more efficient than any deterministic algorithm for solving the problem in dimension $D > 1$. The data structure can be modified to run in $O(\log^2 n)$ expected time per update in the algebraic computation tree model of computation. Even this version is more efficient than the currently best known deterministic algorithm for $D > 2$.

1 Introduction

We consider the *dynamic closest-pair problem*: We are given a set S of n points in D-dimensional space (we assume D 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 $1 \leq t \leq \infty$.

In the L_t-metric, the distance $d_t(p,q)$ between two points $p = (p^{(1)}, \ldots, p^{(D)})$ and $q = (q^{(1)}, \ldots, q^{(D)})$ in D-dimensional space is defined by

$$d_t(p,q) := \left(\sum_{i=1}^{k} |p^{(i)} - q^{(i)}|^t \right)^{1/t},$$

if $1 \leq t < \infty$, and for $t = \infty$, it is defined by

$$d_\infty(p,q) := \max_{1 \leq i \leq D} |p^{(i)} - q^{(i)}|.$$

Throughout this paper, we fix t and measure all distances in the L_t-metric. We write $d(p,q)$ for $d_t(p,q)$.

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 [SH75] and Bentley and Shamos [BS76] 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 computation tree model of computation [Ben83]. 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 [Rab76] on randomized algorithms, gave an $O(n)$ expected time algorithm for this problem. Since then, simpler methods with

*This research was supported by the European Community, Esprit Basic Research Action Number 7141 (ALCOM II). A preliminary version appears in the Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 1993. 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.

†Hongkong UST, Clear Water Bay, Kowloon, Hongkong. email: `golin@cs.ust.hk`. This author was also supported by NSF grant CCR-8918152.

‡UMIACS, University of Maryland, College Park, MD 20742, USA. email: `raman@umiacs.umd.edu`.

§International Computer Science Institute, Berkeley, CA 94704, USA. email: `schwarz@icsi.berkeley.edu`.

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

the same running time have been discovered. Besides the randomized incremental algorithm presented in [GRSS93], there is a different approach, described by Khuller and Matias [KM91], which uses a randomized "filtering" procedure. This method 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 [SSS94]. 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 [Sup90] gave a data structure with $O(\log^D n)$ amortized update time that uses $O(n \log^{D-1} n)$ space. When both insertions and deletions are allowed, Smid [Smi92] described a data structure that uses $O(n \log^D n)$ space and runs in $O(\log^D n \log\log n)$ amortized time per update. Another data structure due to Smid [Smi91], with improvements stemming from results of Salowe [Sal92] and Dickerson and Drysdale [DD91], uses $O(n)$ space and requires $O(\sqrt{n}\log n)$ time for updates. Very recently, after a preliminary version of this paper was presented, Kapoor and Smid [KS94] devised a deterministic data structure of linear size that achieves polylogarithmic amortized update time, namely $O(\log^{D-1} n \log\log n)$ for $D \geq 3$ and $O(\log^2 n/(\log\log n)^\ell)$ for the planar case $D = 2$, where ℓ is an arbitrary non-negative integer constant.

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. It is also assumed that the adversary has no clock to time the algorithm. (This means that, while the adversary knows the algorithm and can choose the input arbitrarily, it cannot derive any information from the performance of the algorithm. See [LN89] for a discussion of clocked adversaries.) We can, however, allow a clocked adversary at the cost of making the updating time bound amortized.

The $O(\log n)$ time 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, we can use ordinary balanced search trees instead, and the update algorithms run in $O(\log^2 n)$ expected time. If we remove both assumptions, we can modify the algorithm to obtain an $O(\log^2 n)$ expected update time in the algebraic computation tree model. Even this version of the randomized algorithm is more efficient than the currently best known deterministic algorithms for solving the problem for $D > 2$, and almost matches the running time of the recently developed method of Kapoor and Smid [KS94] in the planar case $D = 2$, and 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 a randomly chosen one from 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.

The paper is organized as follows. In Section 2, we define the sparse partition, give a static algorithm to build it, and show how to augment this data structure to find the closest pair in constant time.

Then, in Section 3, we show how to implement the sparse partition using a grid data structure. In Section 4, we give the update algorithms and their analysis. In Section 5, we show how to modify the grid data structure in order to obtain an algorithm that fits in the algebraic computation tree model.

In Section 6, we give extensions of the method. We show how to modify the data structure such that we achieve $O(n)$ worst case space rather that only expected, at the cost of making the update time bound amortized. We also give high probability bounds for the running time of a sequence of update operations.

In Section 7, we give some concluding remarks.

2 Sparse partitions

We start with some notation. Let S be a set of n points in D-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 $d(p, S) := \min\{d(p,q) : q \in S \setminus \{p\}\}$. Note that, for convenience, we often speak of *the* closest pair although there might be more than one. This causes no problems since, as we shall see, our data structure not only allows to find a single closest pair, but also to report all pairs of points attaining the minimal distance in time proportional to their number.

As mentioned earlier, the sparse partition is based on the filtering algorithm in [KM91]. In this section, we present an abstract framework that captures all the properties that were needed for that algorithm, along with some modifications that we use for our dynamic data structure. It is convenient to prove a number of results in this framework.

Definition 1 Let S be a set of points in D-space. A *sparse partition* for the set S is a sequence of 5-tuples $(S_i, S_i', p_i, q_i, \delta_i)$, $1 \leq i \leq L$, 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) $\delta_i = d(p_i, q_i) = d(p_i, S_i)$.

(b) For all $1 \leq i \leq L$, and for all $p \in S_i$:

- (b.1) If $d(p, S_i) > \delta_i/2$ then $p \in S_i'$;
- (b.2) If $d(p, S_i) \leq \delta_i/4D$ then $p \notin S_i'$.

(c) For all $1 \leq i < L$, and for all $p \in S_i$:
If $p \in S_{i+1}$, then there is a point $q \in S_i$ such that $d(p,q) \leq \delta_i/2$ and $q \in S_{i+1}$.

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

For each i, we call the points of S_i' the *sparse points in* S_i, and the set S_i' the *sparse set*. Each 5-tuple itself is also called a *level* of the partition.

Conditions (b.1) and (b.2) govern the decision on whether a point of S_i is in the sparse set S_i' or not. The threshold values given in (b.1) and (b.2) depend on the nearest neighbor distance $d(p_i, S_i)$ of the point $p_i \in S_i$, which will be called the *pivot* in the following. For a point $p \in S_i$ such that $d(p_i, S_i)/4D < d(p, S_i) \leq d(p_i, S_i)$, the decision may be arbitrary as long as the results of this section are concerned and will be made precise by the implementation later.

Definition 2 Let $(S_i, S_i', p_i, q_i, \delta_i)$, $1 \leq i \leq L$, be a sparse partition for the set S. A 5-tuple $(S_i, S_i', p_i, q_i, \delta_i)$ is called *uniform* if, for all $p \in S_i$, $\Pr[p = p_i] = 1/|S_i|$. The set S is said to be *uniformly stored* by the sparse partition if all its 5-tuples are uniform.

Lemma 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, $S_1' \cup S_2' \cup \ldots \cup S_L'$ is a partition of S.*

(2) *For any $1 \leq i < L$, $\delta_{i+1} \leq \delta_i/2$. Moreover, $\delta_L/4D < \delta(S) \leq \delta_L$.*

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 $d(p_i, S_i) = \delta_i > \delta_i/2$, it follows from Condition (b.1) in Definition 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 1 that there is a point $q \in S_i$ such that $d(p_{i+1}, q) \leq \delta_i/2$ and $q \in S_{i+1}$. Therefore,

$$\delta_{i+1} = d(p_{i+1}, S_{i+1}) \leq d(p_{i+1}, q) \leq \delta_i/2.$$

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 1 implies that $d(p,S_i) > \delta_i/4D$, and from the first part of (2), we conclude that $\delta(S) > d_i/4D \geq \delta_L/4D$. The inequality $\delta(S) \leq \delta_L$ obviously holds, because δ_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 *Sparse_Partition*(S):

(i) $S_1 := S$; $i := 1$.

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

(iii) Choose S'_i to satisfy (b.1), (b.2) and (c) in Definition 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 *Let S be a set of n points in $\mathbb{R}^D$. Run Sparse_Partition(S) and let $(S_i, S'_i, p_i, q_i, \delta_i)$, $1 \leq i \leq L$, be the 5-tuples constructed by the algorithm. Then this set of 5-tuples is a uniform sparse partition for S, and we have $\mathrm{E}(\sum_{i=1}^{L} |S_i|) \leq 2n$.*

Proof: The output generated by algorithm *Sparse_Partition*(S) obviously fulfills the requirements of Definitions 1 and 2. To prove the bound on the size of the structure, we first note that $L \leq n$ by Lemma 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 $s_{i+1} \leq s_i/2$, from which it follows 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 $d(r, S_i) \geq \delta_i$. Then, Condition (b.1) of Definition 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 $s_{i+1} = \sum_l \mathrm{E}(|S_{i+1}| \mid |S_i| = l) \cdot \Pr(|S_i| = l) \leq s_i/2$. ∎

Let us remark here that the algorithm *Sparse_Partition* is essentially the randomized static closest pair algorithm in [KM91]. That algorithm was only concerned with finding δ_L since with it, one can find the closest pair in $O(n)$ time (see [KM91] for details). Thus, the above algorithm was used as a filtering procedure, it did not have to save the sets S_i and S'_i at all the levels. Our dynamic algorithm will use this information. Particularly, we will now show how the sparse sets S'_i, i.e. the points that were thrown away in each step of the iteration in [KM91], are related to the minimal distance $\delta(S)$. This relation will help us to use the sparse partition to find $\delta(S)$ quickly, and will be the core of our dynamic data structure for maintaining the closest pair.

Definition 3 Let $S'_1, S'_2, \ldots, S'_L$ be the sparse sets of a sparse partition for S. For any $p \in \mathbb{R}^D$ and $1 \leq i \leq L$, define the *restricted distance*

$$d^*_i(p) := \min\Big(\delta_i,\ d\big(p, \bigcup_{1 \leq j \leq i} S'_j\big)\Big),$$

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

For convenience, we define, for all $i \leq 0$, $S'_i := \emptyset$, $\delta_i := \infty$, and $d^*_i(p) := \infty$ for any point p.

Lemma 3 *Let $p \in S$ and let i be the index such that $p \in S'_i$.*

(1) *$d^*_i(p) > \delta_i/4D$.*

(2) *If $q \in S'_j$, where $1 \leq j < i - D$, then $d(p,q) > \delta_i$.*

(3) *$d^*_i(p) = \min\left(\delta_i,\ d(p, S'_{i-D} \cup S'_{i-D+1} \cup \ldots \cup S'_i)\right)$.*

Proof: (1) Let $1 \le j \le i$ and let $q \in S'_j$. Since $p \in S_j$, it follows from Condition (b.2) of Definition 1 that $d(p,q) \ge d(q, S_j) > \delta_j/4D \ge \delta_i/4D$.

(2) Let $q \in S'_j$, where $1 \le j < i - D$. As in (1), we get $d(p,q) > \delta_j/4D$. Then, Lemma 1 implies that

$$d(p,q) > \frac{\delta_j}{4D} \ge \frac{\delta_{i-D-1}}{4D} \ge \frac{2^{D+1}\delta_i}{4D} \ge \delta_i.$$

(3) follows immediately from (2). ∎

Lemma 4

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

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

It remains to prove that we can restrict the value of i to $L - D, L - D + 1, \ldots, L$. We know from Lemma 3 (1) that $\min_{p \in S'_i} d_i^*(p) > \delta_i/4D$. Moreover, we know from Lemma 1 (2), that for $i < L - D$, $\delta_i/4D \ge \delta_{L-D-1}/4D \ge (2^{D+1}/4D) \cdot \delta_L \ge \delta_L \ge \delta(S)$. ∎

Now we are ready to describe how to find the closest pair using the sparse partition. According to the characterization of $\delta(S)$ in Lemma 4, we will augment the sparse partition with a data structure that stores, for each level $i \in \{1, \ldots, L\}$, the set of restricted distances $\{d_i^*(p) : p \in S'_i\}$.

The data structure that we use for this purpose is called *heap*. Heaps are described, for example, in [CLR90, Chapter 7]. A heap is a data structure that stores a set of *items*, ordered by a real-valued *key*. The items are stored in a binary tree that satisfies the following *partial ordering:* the keys of the children of a node are not smaller than the key of the node itself. It follows that the minimum key is stored at the root of the tree. This heap variant is called *min-heap*. The operations that can be carried out on a heap, together with their running times, are summarized in the following lemma:

Lemma 5 *Let I be a set of n items. Then a heap H storing these n items can be built in linear time, and the following operations are supported in $O(\log n)$ time:*

- *insert*(item) *and delete*(item),
- *change_key*(item, key), *changing the key of the item* item *in H to the value* key.

For the operations dealing with an item in the heap H, namely delete *and* change_key, *access to this item is assumed to be available by a pointer.*

Operation find_min(H), returning an item with minimum key in H, can be performed in $O(1)$ time, and find_all_min(H), which returns all A items with minimum key in H, can be performed in time $O(A)$.

Proof: For building of the heap and the operations *insert* and *delete*, see [CLR90]. *Change_key* can be implemented by *insert* and *delete*. Operation *find_min*(H) simply returns the item stored at the root of the tree that represents the heap H, as mentioned above.

Moreover, we can find all items whose key is equal to the minimum key in time proportional to their number, by performing a depth first search in the binary tree, stopping when a node containing a key that is larger than the minimum key is encountered. Thus, the total number of nodes inspected is at most twice the number of items reported. ∎

So, for each $i \in \{1, \ldots, L\}$, we maintain a min-heap H_i that stores items having the restricted distances $\{d_i^*(p) : p \in S'_i\}$ as their keys. How we compute these values depends on the way we implement the sparse partition, which will be described in the next sections. There, we also describe the exact contents of our heap items.

Lemma 6 *Let $S'_1, S'_2, \ldots, S'_L$ be the sparse sets of a sparse partition for S, and for each $1 \leq i \leq L$, let the set $\{d^*_i(p) : p \in S'_i\}$ of restricted distances be stored in a min-heap H_i. Then the minimum distance $\delta(S)$ can be found in constant time. Moreover, all point pairs attaining this minimum distance can be reported in time proportional to their number.*

Proof: For $i \leq 0$, define H_i as the empty heap. Lemma 4 characterizes $\delta(S)$ as a minimum of certain restricted distances. In particular, Lemma 4 says that $\delta(S)$ can only be stored in one of the heaps $H_{L-D}, H_{L-D+1}, \ldots, H_L$. To find $\delta(S)$ it is therefore enough to take the minima of these $D+1$ heaps and then to take the minimum of these $D+1$ values. Moreover, we can report all closest pairs in time proportional to their number, as follows: in all of the at most $D+1$ heaps whose minimum key is $\delta(S)$, we report all items whose key is equal to $\delta(S)$. From the discussion of heaps above, this can be done in time proportional to the number of items that are reported. ∎

We close this section with an abstract description of our data structure.

The closest-pair data structure for set S:

- A data structure storing S uniformly as a sparse partition according to Definitions 1 and 2.
- The heaps $H_1, H_2, \ldots, H_L$, where H_i stores the set of restricted distances $d^*_i(p)$, cf. Definition 3, for all points p in the sparse set S'_i.

In the rest of the paper,we discuss two 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 variant of the data structure. Concerning implementation details and update algorithms, we then only mention the changes that have to be made in comparison to the grid based implementation in order to establish the results.

3 Implementation of the sparse partition

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

3.1 The notion of neighborhood in grids

We start with some definitions. Let $\delta > 0$. We use $\mathcal{G}_\delta$ to denote the grid with mesh size δ and a lattice point at $(0, 0, \ldots, 0)$. Hypercubes of the grid are called *boxes*. More precisely, a box has the form

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

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 $\mathbb{R}^D$ is contained in exactly one grid box.

The neighborhood of a box b in the grid $\mathcal{G}_\delta$, denoted by $N(b)$, consists of b itself plus the collection of $3^D - 1$ boxes bordering on it.

Let p be any point in $\mathbb{R}^D$ and let $b_\delta(p)$ denote the box of $\mathcal{G}_\delta$ that contains p. The *neighborhood of p in $\mathcal{G}_\delta$*, denoted by $N_\delta(p)$, is defined as the neighborhood of $b_\delta(p)$, i.e. $N_\delta(p) := N(b_\delta(p))$.

Now we consider the neighborhood of a point $p \in \mathbb{R}^D$ restricted to a set of points. Let $\mathcal{G}_\delta$ be a grid. Let V be a set of points in $\mathbb{R}^D$. The *neighborhood of p in $\mathcal{G}_\delta$ relative to V*, is defined as

$$N_\delta(p, V) := N_\delta(p) \cap (V \setminus \{p\}).$$

We say that *p is sparse in $\mathcal{G}_\delta$ relative to V* if $N_\delta(p, V) = \emptyset$, i.e. if, besides p, there are no points of V in $N_\delta(p)$. In cases that V and δ are understood from the context we will simply say that p is *sparse.*

Using these notations, we immediately get

Fact 7 *Let V be a set of points in $\mathbb{R}^D$, and let p and q be arbitrary points in $\mathbb{R}^D$.*

(N.1) *If* $N_\delta(p,V)=\emptyset$, *then* $d(p,V)>\delta$.

(N.2) *If* $q\in N_\delta(p,V)$, *then* $d(p,q)\le 2D\delta$

(N.3) $q\in N_\delta(p)\Longleftrightarrow p\in N_\delta(q)$.

We are now in a position to define the sets S_i' precisely. For $i\ge 1$, let

$$S_i' := \{p\in S_i\ :\ p \text{ sparse in } \mathcal{G}_{\delta_i/4D} \text{ relative to } S_i\}. \tag{1}$$

For convenience, let us rewrite the abstract definition of the sparse partition given in Definition 1. We simply replace the conditions that concern S_i' in Definition 1, namely (b.1), (b.2) and (c), with the definition of Equation (1). With the new definition, we will be able to specify and analyze our update algorithms in Section 4.

Definition 4 A sparse partition for the set S is a sequence of 5-tuples $(S_i,S_i',p_i,q_i,\delta_i)$, $1\le i\le L$, where L is a positive integer, such that:

1. For $i=1,\dots,L$:
   (a) $\emptyset\ne S_i\subseteq S$;
   (b) $p_i,q_i\in S_i$ and $p_i\ne q_i$ if $|S_i|>1$;
   (c) $\delta_i=d(p_i,q_i)=d(p_i,S_i)$.
   (d) $S_i'=\{p\in S_i\ :\ N_{\delta_i/4D}(p,S_i)=\emptyset\}$.
2. $S_1=S$ and for $1\le i\le L-1$, $S_{i+1}=S_i\setminus S_i'$.

Lemma 8 *Let* $(S_i,S_i',p_i,q_i,\delta_i)$, $1\le i\le L$, *be a set of 5-tuples satisfying Definition 4. Then this set of 5-tuples also satisfies Definition 1.*

Proof: We only have to prove Conditions (b) and (c) of Definition 1. Let $1\le i\le L$ and let $p\in S_i$. First assume that $p\notin S_i'$. Then, there is a point $q\in S_i$ that is in the neighborhood of p. By (N.2), $d(p,S_i)\le d(p,q)\le 2D\cdot\delta_i/4D=\delta_i/2$. This proves Condition (b.1). To prove (b.2), assume that $p\in S_i'$. Then, the neighborhood of p relative to S_i is empty. Hence, by (N.1), $d(p,S_i)>\delta_i/4D$.

To prove (c), let $1\le i<L$ and let $p\in S_{i+1}=S_i\setminus S_i'$. It follows that there is a point $q\in S_i$ such that $q\in N_{\delta_i/4D}(p)$. By the symmetry property (N.3), this is equivalent to $p\in N_{\delta_i/4D}(q)$ and therefore $q\in S_{i+1}$. From Condition (b.1), we also have $d(p,q)\le\delta_i/2$. ∎

We now come to some additional properties of the sparse partition as defined in Definition 4 that will be used for the dynamic maintenance of the data structure. For this purpose, we give some additional facts about neighborhoods.

We start with some notation. Let p be a point in $\mathbb{R}^D$. We number the 3^D boxes in the neighborhood of p as follows. The number of a box is a D-tuple over $\{-1,0,1\}$. The j-th component of the D-tuple is -1, 0, or 1, depending on whether the j-th coordinate of the box (i.e. its lower left coordinate) is smaller than, equal to or greater than the corresponding coordinate of $b_\delta(p)$. We call this D-tuple the *signature* of a box. We denote by $b_\delta^\Psi(p)$ the box with signature Ψ in $N_\delta(p)$.

We are now going to define the notion of *partial neighborhood* of a point p. See Figure 1. For any signature Ψ, we denote by $N_\delta^\Psi(p)$ the part of p's neighborhood that is in the neighborhood of $b_\delta^\Psi(p)$. Note that $N_\delta^\Psi(p)$ contains all the boxes $b_\delta^{\Psi'}(p)$ of $N_\delta(p)$ whose signature Ψ' differs from Ψ by at most 1 for each coordinate—these are exactly the boxes bordering on $b_\delta^\Psi(p)$ including $b_\delta^\Psi(p)$ itself. Particularly, $N_\delta^{0,\dots,0}(p)=N_\delta(p)$, i.e. the partial neighborhood with signature $0,\dots,0$ is the whole neighborhood of p.

The following properties relate the neighborhoods of different grids.

Lemma 9 *Let* $0<\delta'\le\delta''/2$ *be real numbers and let* $p\in\mathbb{R}^D$. *Then*

(N.4) $N_{\delta'}(p)\subseteq N_{\delta''}(p)$.

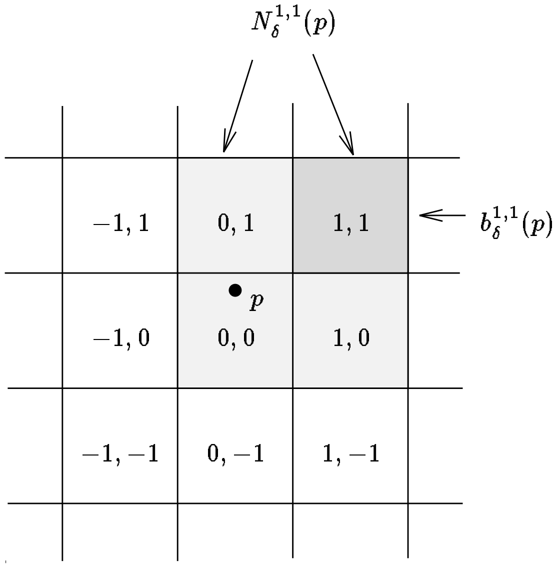

Figure 1: The neighborhood of a point p in $\mathcal{G}_\delta$. The dark shaded area denotes the box $b_\delta^{1,1}(p)$ in the upper right corner of p's neighborhood. This box also belongs to $N_\delta^{1,1}(p)$, the partial neighborhood of p with signature $1, 1$. The light shaded area shows the other three boxes of $N_\delta^{1,1}(p)$.

(N.5) *For any signature* $\Psi \in \{-1, 0, 1\}^D$*:* $b_{\delta'}^{\Psi}(p) \subseteq N_{\delta''}^{\Psi}(p)$.

Proof: For any grid size δ and $1 \leq j \leq D$, denote by $h_\delta^{L,j}, h_\delta^{l,j}, h_\delta^{r,j}, h_\delta^{R,j}$ the j-th coordinates of the four hyperplanes bounding the grid boxes of p's neighborhood in the j-direction, ordered from "left" to "right". See Figure 2.

Let $q = (q^{(1)}, q^{(2)}, \ldots, q^{(D)}) \in \mathbb{R}^D$, and let $\Psi = (\alpha_1, \ldots, \alpha_D) \in \{-1, 0, 1\}^D$ be a signature. Then $q \in b_\delta^{\Psi}(p)$ in $\mathcal{G}_\delta$ if and only if, for all $1 \leq j \leq D$:

$$\begin{array}{ll} h_\delta^{l,j} \leq q^{(j)} \leq h_\delta^{r,j} & \text{if } \alpha_j = 0 \\ h_\delta^{r,j} \leq q^{(j)} \leq h_\delta^{R,j} & \text{if } \alpha_j = 1 \\ h_\delta^{L,j} \leq q^{(j)} \leq h_\delta^{l,j} & \text{if } \alpha_j = -1 \end{array}$$

Also, $q \in N_\delta^{\Psi}(p)$ if and only if, for all $1 \leq j \leq D$:

$$\begin{array}{ll} h_\delta^{L,j} \leq q^{(j)} \leq h_\delta^{R,j} & \text{if } \alpha_j = 0 \\ h_\delta^{l,j} \leq q^{(j)} \leq h_\delta^{R,j} & \text{if } \alpha_j = 1 \\ h_\delta^{L,j} \leq q^{(j)} \leq h_\delta^{r,j} & \text{if } \alpha_j = -1 \end{array}$$

Figure 2 shows the neighborhoods of p in the two grids $\mathcal{G}_{\delta'}$ and $\mathcal{G}_{\delta''}$.

Now observe that, since $\delta' \leq \delta''/2$,

$$h_{\delta'}^{L,j} \geq h_{\delta''}^{L,j} \tag{2}$$

$$h_{\delta'}^{R,j} \leq h_{\delta''}^{R,j} \tag{3}$$

for $1 \leq j \leq D$.

These facts are equivalent to $N_{\delta'}(p) \subseteq N_{\delta''}(p)$, which is claim (N.4).

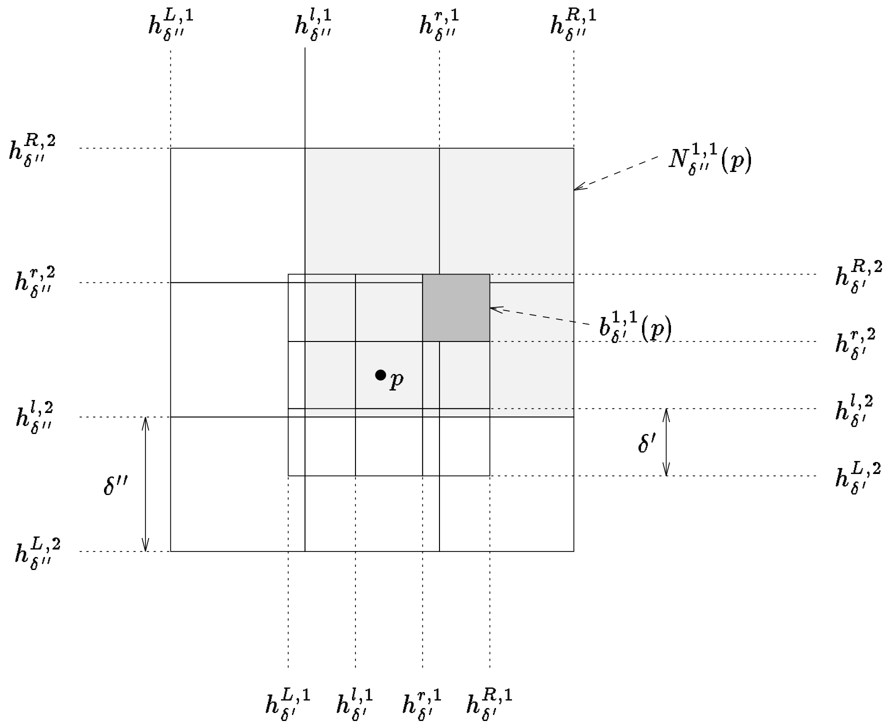

Figure 2: The neighborhoods of a point p in grids $\mathcal{G}_{\delta'}$ and $\mathcal{G}_{\delta''}$ where $\delta' \leq \delta''/2$.

Furthermore, by the definition of the hyperplanes w.r.t. p,

$$h^{r,j}_{\delta'} \geq h^{l,j}_{\delta''} \tag{4}$$

$$h^{l,j}_{\delta'} \leq h^{r,j}_{\delta''} \tag{5}$$

for $1 \leq j \leq D$.
This proves claim (N.5): $b^{\Psi}_{\delta'}(p) \subseteq N^{\Psi}_{\delta''}(p)$. ∎

Notation: Consider a set S that is stored in a set of 5-tuples $(S_i, S'_i, p_i, q_i, \delta_i)$, $1 \leq i \leq L$, according to Definition 4. Since we will only use grids $\mathcal{G}_{\delta_i/4D}$ for the data structures that store level i of the partition, we will use the short notations $\mathcal{G}_i := \mathcal{G}_{\delta_i/4D}$ and $N_i(p) := N_{\delta_i/4D}(p)$ from now on. We use the same convention for the neighborhood relative to a set.

Corollary 1 *Let p be an arbitrary point of $\mathbb{R}^D$, and let $(S_i, S'_i, p_i, q_i, \delta_i)$, $1 \leq i \leq L$, be a sparse partition. Then, for any $1 \leq i < j \leq L$,*

$$N_j(p) \subseteq N_i(p).$$

Proof: Apply (N.4) from Lemma 9 with $\delta'' = \delta_i/4D, \delta' = \delta_j/4D$, noting that $\delta' \leq \delta''/2$ by Lemma 1. ∎

Particularly, if $N_i(p, S_i) = \emptyset$ for a point $p \in \mathbb{R}^D$, i.e. if p is sparse in $\mathcal{G}_i$ relative to S_i, then, since $S_{i+1} \subseteq S_i$, Corollary 1 implies $N_{i+1}(p, S_{i+1}) = \emptyset$, which means that p is also sparse in $\mathcal{G}_{i+1}$ relative to S_{i+1}. This property will be crucial to our update algorithms. The following lemma will also be useful later on.

Lemma 10 *Let $(S_i, S'_i, p_i, q_i, \delta_i)$, $1 \leq i \leq L$, be a sparse partition. Then, for any $p \in S \setminus S_{i+1}, 1 \leq i < L$,*

$$N_i(p, S) = \emptyset.$$

Operation	Time
$V \cap b = \emptyset$?	$O(1)$
report $V \cap b$	$O(\|V \cap b\|)$
q sparse in $\mathcal{G}_\delta$ relative to V ?	$O(1)$

Table 1: V is stored according to $\mathcal{G}_\delta$, b is a box of $\mathcal{G}_\delta$, and q is an arbitrary point.

Proof: We use induction on i. If $i = 1$, then for any $p \in S \setminus S_2 = S_1'$, $N_1(p, S) = N_1(p, S_1) = \emptyset$ by definition. Now let $i > 1$ and assume that, for any $p \in S \setminus S_i$, we have $N_{i-1}(p, S) = \emptyset$.

1. If $p \in S \setminus S_i$, then $N_i(p, S) \subseteq N_{i-1}(p, S) = \emptyset$ by Corollary 1 and our induction hypothesis, respectively.
2. If $p \in S_i'$, then $N_i(p, S_i) = \emptyset$ by definition. It remains to show that $N_i(p, S \setminus S_i) = \emptyset$. This is true because if there were a point $q \in S \setminus S_i$ such that $q \in N_i(p)$, then by the symmetry property (N.3), we had $p \in N_i(q)$, contradicting $N_i(q, S) = \emptyset$, which was shown in item 1 above.

We have thus shown that $N_i(p, S) = \emptyset$ for any $p \in S \setminus S_{i+1} = (S \setminus S_i) \cup S_i'$. ∎

3.2 Storing a point set according to a grid

Having defined the grid based implementation by specifying the sets S_i', we can now explain how to store the point sets involved in the sparse partition of our input set S. Let d be a grid size and $V \subseteq S$ a subset of S. Then we use perfect hashing (see [DM90, FKS84]) to store the points of V.

Particularly, the dynamic hashing scheme of [DM90] allows to store a set of integer-valued keys, for which a bound on their size is known in advance, in linear space such that the information stored at a given key can be accessed in $O(1)$ worst case time, and a key can be inserted or deleted in $O(1)$ expected time. (The time bound is even attained with high probability, but we do not need this in our application.)

For each point, we take as a key the index of the box in $\mathcal{G}_\delta$ 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 $V \cap b$, in arbitrary order. We call this *storing V according to $\mathcal{G}_\delta$*, and the data structure itself is called the *box dictionary*. Note that the size of the box indices must be bounded to make possible the use of hashing. As mentioned earlier, we assume the prior knowledge of a frame containing all the points for this version of the algorithm. This bound, together with a given δ, gives a bound on the size of the box indices *before* we build a data structure for $\mathcal{G}_\delta$.

If V is stored according to $\mathcal{G}_\delta$, then we can answer the question *"are any points of V in box b?"* in $O(1)$ worst case time. Moreover, if the answer is yes, we can report all points in $V \cap b$ in time proportional to their number. By checking all boxes in the neighborhood of an arbitrary point q, we can check in $O(1)$ time if q is sparse in the grid $\mathcal{G}_\delta$ relative to V. Therefore, by doing this for each point in V we can, in linear time, find the subset $V' \subseteq V$ of sparse points in V.

Table 1 summarizes the basic operations on a (not necessarily sparse) point set V stored according to a grid $\mathcal{G}_\delta$.

3.3 The complete data structure

Recall that, when discussing a sparse partition, we use $\mathcal{G}_i$ as a short form for the grid of mesh size $\delta_i/4D$. Our data structure now consists of the following:

For each $1 \leq i \leq L$:

- the pivot $p_i \in S_i$, its nearest neighbor q_i in S_i and $\delta_i = d(p_i, q_i)$,
- S_i stored according to $\mathcal{G}_i$.

- S_i' stored according to $\mathcal{G}_i$.
- the heap H_i.

Note that this means that S_i and S_i' are kept in two separate grid data structures defined on $\mathcal{G}_i$. We now add some more details to the description. Let b be a box of $\mathcal{G}_i$ that is non-empty w.r.t. to S_i or S_i'. The list of points in $S_i \cap b$ will be called $\mathcal{L}(b)$, and the list of points in $S_i' \cap b$ will be called $\mathcal{L}'(b)$. Each element of $\mathcal{L}(b)$ is a record containing the following information:

$p \in \mathcal{L}(b)$: record	*point: p*	*upper:* $\uparrow p$ in S_{i-1}	*lower* : $\uparrow p$ in S_{i+1}

Here, "$\uparrow p$ in V" means a pointer to the representation of point p in the data structure storing V. The pointers are nil if the corresponding representation of the point does not exist.

Each element of $\mathcal{L}'(b)$ is a record with the following information:

$p \in \mathcal{L}'(b)$: record	*point: p*	*it* : $\uparrow it(p)$ in H_i	*left* : $\uparrow p$ in S_i

Here, "$\uparrow it(p)$ in H_i" means a pointer to the heap item $it(p)$ with key $d_i^*(p)$, see below. Note that each list $\mathcal{L}'(b)$ actually contains at most one point, by the sparseness property of the set S_i'. We use this general data structure in order to have a simple description *and* because, during the update operations, we will face intermediate stages where we temporarily insert points into the data structure of a sparse set that make the set non-sparse.

Now let us turn to the heaps. The heaps contain items, ordered by a real-valued key. For each key, there is some associated information. The key of an item in heap H_i is the value $d_i^*(p)$ for some $p \in S_i'$. Let q be such that $d_i^*(p) = d(p,q) < \delta_i$, and let l be such that $0 \leq l \leq D$ and $q \in S_{i-l}'$. Then the heap item $it(p)$ of H_i contains the following information:

item $it(p) \in H_i$: record	*key:* $d_i^*(p)$	*point:* $\uparrow p$ in S_i'	*point2:* $\uparrow q$ in S_{i-l}'

If the point q does not exist, i.e. if $d_i^*(p) = \delta_i$, then the pointer *point2* is nil.

Remark: The pointers described above have the following reason: the elements in the data structures for the non-sparse sets S_i are linked to facilitate the deletion of a point from the data structure. The other pointers are connecting the heaps with the data structures that store the sparse partition. They will be needed when we maintain the heaps during update operations. ∎

We can now give the algorithm to build the complete data structure. We fill in the implementation details of the algorithm *Sparse_Partition* given in Section 2, and we also construct the heaps.

Algorithm *Build*(S)

1. $S_1 := S$; $i := 1$.
2. Choose a random point $p_i \in S_i$. Calculate $\delta_i := d(p_i, S_i)$. Let $q_i \in S_i$ be such that $d(p_i, q_i) = \delta_i$.
3. Store S_i according to $\mathcal{G}_i$.
4. Compute $S_i' := \{p \in S_i \ : \ p \text{ sparse in } \mathcal{G}_i \text{ relative to } S_i\}$.
5. Store S_i' according to $\mathcal{G}_i$.
6. Compute the restricted distances $\{d_i^*(p) \ : \ p \in S_i'\}$ and, using a linear time algorithm, construct a heap H_i containing these values with the minimal value at the top.
7. If $S_i = S_i'$ stop; otherwise set $S_{i+1} := S_i \setminus S_i'$, set $i := i+1$ and goto 2.

For the sake of simplicity, we did not mention in this algorithm how to establish the above described links between the various parts of the data structure. The links between heap items and points in a list $\mathcal{L}'(b)$, i.e. points stored in a sparse set S_i', can be installed during the construction of the heaps. The pointers between representations of a point p in subsequent non-sparse sets S_i, S_{i+1} can be easily established in step 7, when S_{i+1} is obtained by stripping off the sparse set S_i' from S_i.

Lemma 11 *Let $(S_i, S_i', p_i, q_i, \delta_i), 1 \leq i \leq L$, be a sparse partition, and let $p \in S_i'$ for some $i \in \{1, \ldots, L\}$. If we have the data structures storing the sets S_j' according to $\mathcal{G}_j$ available for $1 \leq j \leq i$, then the value $d_i^*(p)$ can be computed in $O(1)$ time.*

Proof: We know from Lemma 3 (2) that if $d_i^*(p) = d(p,q)$ with $d(p,q) < \delta_i$ then q must be in one of the sets $S_i', S_{i-1}', \ldots, S_{i-D}'$. Furthermore, there are only a constant number of boxes in the grids $\mathcal{G}_j$, $i - D \leq j \leq i$, where the point q can possibly appear in: since the boxes in the grids $\mathcal{G}_j$ have side length $\delta_j/4D \geq \delta_i/4D$, these are the grid boxes that are within $4D$ boxes of the box that p is located in. Finally, because of the sparseness of the sets S_j', there can be at most one point found in each grid box. Therefore, using the hash tables storing S_j', $i - D \leq j \leq i$, we can find all points contained in these boxes and compute $d_i^*(p)$ in $O(1)$ time. ∎

Lemma 12 *Given a set S of n points in D-space, a sparse partition for S according to Definition 4 can be built in $O(n)$ expected time and has $O(n)$ expected size.*

Proof: Consider the i-th iteration of algorithm *Build*(S). Step 2 can be performed in $O(|S_i|)$ deterministic time by calculating the distance between p_i and all other points in S_i. Steps 3 and 5 build the grid data structures for S_i and S_i' and take $O(|S_i|)$ and $O(|S_i'|)$ expected time, respectively. By the discussion at the end of Subsection 3.2, step 4, which computes S_i' from S_i, can be performed in $O(|S_i|)$ deterministic time. This implicitly includes the work of step 7.

Since we have the data structures for S_j', $1 \leq j \leq i$, available in the i-th iteration of the algorithm, we can apply Lemma 11 to conclude that computing the restricted distances $\{d_i^*(p) : p \in S_i'\}$ in step 6 takes $O(|S_i'|)$ worst case time. The heap H_i can be constructed within the same time bound.

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 shows that this quantity is $O(n)$. ∎

Recall that given this data structure, we can find the closest pair in S in $O(1)$ time by Lemma 6.

4 Dynamic maintenance of the data structure

In this section, we show how to maintain the sparse partition when the input set S is modified by insertions and deletions of points. The algorithms for insertions and deletions turn out to be very similar. We will demonstrate the ideas that are common to both update operations when we treat insertions. Next, we give the deletion algorithm.

4.1 The insertion algorithm

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 (the pivot of S_1) is a random element of $S_1 = S$. 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—the nearest neighbor of p_1—and, hence, δ_1 have to be changed. First note that q can be the nearest neighbor of at most $3^D - 1 \leq 3^D$ points in S_1. (See [DE84].) This means that δ_1 changes only if p_1 is one of these points. Since we assumed that the adversary cannot see the coin flips of the algorithm, and since p_1 is chosen uniformly from S_1, it follows that the probability of δ_1 changing is at most $3^D/|S_1|$. If δ_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 δ_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 $(S_i, S'_i, p_i, q_i, \delta_i)$ and $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{\delta}_i)$ are identical for $2 \le i \le L$. 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, i.e., we have to construct the 5-tuple for $\widetilde{S}_2$.

Let us now describe the algorithm to construct the new 5-tuple for $\widetilde{S}_i, i \ge 1$. In this process, we will extend the notion of the set $down_1$ from the first level to the other levels of the sparse partition.

We define $down_0 := \emptyset$. Let $i \ge 1$. The following invariant holds if the algorithm attempts to construct the 5-tuple for $\widetilde{S}_i$ without having made a rebuilding yet:

Invariant $\mathsf{INS}(i)$**:**

(a) For $1 \le j < i$:

(a.1) $q \in \widetilde{S}_j$ and the 5-tuple $(\widetilde{S}_j, \widetilde{S}'_j, \widetilde{p}_j, \widetilde{q}_j, \widetilde{\delta}_j)$, satisfies Definitions 4 (sparse partition) and 2 (uniformness), where $\widetilde{p}_j = p_j, \widetilde{q}_j = q_j, \widetilde{\delta}_j = \delta_j$;

(a.2) $\widetilde{S}_{j+1} = \widetilde{S}_j \setminus \widetilde{S}'_j$;

(b) The sets $down_j, 0 \le j < i$, have been computed and $\widetilde{S}_i = S_i \cup down_{i-1} \cup \{q\}$.

Note that at the start of the algorithm, $\mathsf{INS}(1)$ holds because $down_0 = \emptyset$. We will show later that 3^D is an upper bound on the size of the union of all the *down* sets. In particular, each single *down* set has size at most 3^D.

Now let us construct the 5-tuple for $\widetilde{S}_i$. From invariant $\mathsf{INS}(i)$ (b), we have $\widetilde{S}_i = S_i \cup down_{i-1} \cup \{q\}$. As discussed above, to construct the first 5-tuple we had to consider the new point q as new pivot with probability $1/|S_1|$. In general, constructing $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{\delta}_i)$ from $(S_i, S'_i, p_i, q_i, \delta_i)$ requires that, instead of one, up to $3^D + 1$ points (q as well as 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 nearest neighbor, but this only increases the probabilities by a constant factor.

If no rebuilding takes place, we determine $\widetilde{S}'_i$, the set of sparse points in $\widetilde{S}_i$. In order to do this, we define the set $down_i$. In this set, we want to collect the points of S that were already sparse at some level $j \le i$, but that will not be sparse at level i due to the insertion of q. We do this as follows. Let $D_i := S'_i \cup down_{i-1}$. Then

$$down_i := N_i(q, D_i) = \{x \in D_i \,:\, x \in N_i(q)\}\,. \tag{6}$$

The set D_i is called the "candidate set" for $down_i$. We can compute $\widetilde{S}'_i$ as follows: throw out from D_i all elements that belong to $down_i$ and add q, if it is sparse in $\widetilde{S}_i$. (We shall prove later that the set $\widetilde{S}'_i$ computed in this way actually *is* the set of sparse points in $\widetilde{S}_i$.)

We have constructed the 5-tuple $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{\delta}_i)$ and can now compute $\widetilde{S}_{i+1} = \widetilde{S}_i \setminus \widetilde{S}'_i$, the next subset of our new sparse partition for $\widetilde{S}$. If $q \in \widetilde{S}'_i$ then, by the definition of the *down* sets, $down_i = \emptyset$ and $S_{i+1} = \widetilde{S}_{i+1}$. This means that the levels $i+1, \ldots, L$ of the sparse partition remain unchanged, and we are finished with the construction of the sparse partition for $\widetilde{S}$. Otherwise, $q \in \widetilde{S}_{i+1}$. So, q and the points in $down_i$ are not sparse in $\widetilde{S}_i$ and we can add q and $down_i$ to S_{i+1}, giving the set $\widetilde{S}_{i+1}$. The invariant $\mathsf{INS}(i+1)$ holds, as we will prove later. We then continue with level $i+1$.

After the sparse partition has been updated, it remains to update the heaps. It is clear that the new point q has to be inserted into the heap structure appropriately. To see what kind of changes will be performed for the points of S, let us examine the point movements between the different levels of the sparse partition due to the insertion of q more closely. Let us look at level i, where $i \geq 1$. From invariants $\mathsf{INS}(i)$ (b) resp. $\mathsf{INS}(i+1)$ (b), the points in $down_{i-1}$ resp. $down_i$ move at least down to level i resp. level $i+1$. The construction rule for $\widetilde{S}'_i$ now implies $\widetilde{S}'_i \setminus \{q\} = (S'_i \cup down_{i-1}) \setminus down_i$. Thus, we have the following

Fact 13 *Let p be a point in S:*

(i) $p \in down_i \setminus down_{i-1} \Longleftrightarrow p \in S'_i$ *and* $p \notin \widetilde{S}'_i$,

(ii) $p \in down_{i-1} \setminus down_i \Longleftrightarrow p \notin S'_i$ *and* $p \in \widetilde{S}'_i$,

(iii) $p \in down_{i-1} \cap down_i \Longleftrightarrow p \notin S'_i$ *and* $p \notin \widetilde{S}'_i$.

That is, the points in (i) *start moving* at level i, the points in (ii) *stop moving* at level i, and the points in (iii) *move through* level i. For all the points satisfying (i) or (ii), we have to update the heaps where values associated with these points disappear (i) or enter (ii).

The complete insertion algorithm is given in Figure 3. It remains to describe the procedures that actually perform the heap updates. Before we do this, however, we will show that steps **1-3** in lines (3)-(21) of the algorithm actually produce a sparse partition for the new set $\widetilde{S}$.

Lemma 14 *Assume that algorithm Insert(q) has constructed 5-tuples $(\widetilde{S}_j, \widetilde{S}'_j, \widetilde{p}_j, \widetilde{q}_j, \widetilde{d}_j)$, $1 \leq j < i$, and a set $\widetilde{S}_i$, that satisfy $\mathsf{INS}(i)$. Then, if no rebuilding is made, the i-th iteration of the algorithm constructs the 5-tuple $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{\delta}_i)$ and the set $\widetilde{S}_{i+1}$, which satisfy $\mathsf{INS}(i+1)$ (a.1)-(a.2). Furthermore, if $q \notin \widetilde{S}'_i$, then $\mathsf{INS}(i+1)$ (b) also holds.*

Proof: Let us first prove (a.1), saying that the 5-tuple $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{\delta}_i)$ satisfies Definitions 4 and 2, with $\widetilde{p}_i = p_i, \widetilde{q}_i = q_i$ and $\widetilde{\delta}_i = \delta_i$. The 5-tuple is certainly uniform, and it retains the pivot as well as the pivot's nearest neighbor when the algorithm has passed step **2** (check for rebuild) of the algorithm without a rebuilding, cf. the discussion at the beginning of this section.

The fact that the new 5-tuple is still uniform holds because we assume that the adversary has no clock. Note that the uniformness condition $\Pr[p = p_i] = 1/|S_i|$ in Definition 2, concerning the knowledge of the adversary, means that the adversary must consider each element of S_i as possible pivot with equal probability. If we would allow the adversary to time the algorithm, then she could distinguish the events "rebuilding" and "no rebuilding" by their different running times, at least as long as the subset S_i is large enough. From the former event, no information can be gained. In the latter case, however, the adversary would know that all points of $\widetilde{S}_i$ that are nearest neighbor of one of the new points at that level, i.e. the points in $down_{i-1} \cup \{q\}$, cannot be the pivot p_i, and so the 5-tuple is no longer uniform.

Returning to our assumption that the adversary has no clock, it is clear from the above discussion that in this case, the adversary does not notice and hence, cannot derive any information from the event that the insertion algorithm did not rebuild.

It remains to show that $N_i(p, \widetilde{S}_i) = \emptyset \Longleftrightarrow p \in \widetilde{S}'_i$ for any $p \in \widetilde{S}_i$, see Definition 4, 1(d). We have $\widetilde{S}_i = S_{i+1} \cup S'_i \cup down_{i-1} \cup \{q\}$ from invariant $\mathsf{INS}(i)$ (b). Since $N_i(p, \widetilde{S}_i) \neq \emptyset$ for $p \in S_{i+1}$, it remains to prove the claim for $p \in D_i = S'_i \cup down_{i-1}$ and $p = q$. Note that, since $D_i \subseteq S \setminus S_{i+1}$, we have $N_i(p, S) = \emptyset$ by Lemma 10 and therefore, for any $p \in D_i$,

$$\begin{aligned} N_i(p, \widetilde{S}_i) = \emptyset &\Longleftrightarrow q \notin N_i(p) \\ &\Longleftrightarrow p \notin N_i(q) \quad \text{by symmetry (N.3)} \\ &\Longleftrightarrow p \notin down_i \quad \text{by definition of } down_i \\ &\Longleftrightarrow p \in \widetilde{S}'_i \quad \text{by definition of } \widetilde{S}'_i. \end{aligned}$$

If $p = q$, then $N_i(p, \widetilde{S}_i) = \emptyset \Longleftrightarrow q \in \widetilde{S}'_i$ by lines (17)-(19).

Next, we show that $\widetilde{S}_{i+1} = \widetilde{S}_i \setminus \widetilde{S}'_i$. After line (16), we have $\widetilde{S}_{i+1} = S_{i+1} \cup down_i$ and $\widetilde{S}'_i = (S'_i \cup down_{i-1}) \setminus down_i$, and at the end of step **3**, q has been added to exactly one of these sets. Thus $\widetilde{S}_{i+1} \cup \widetilde{S}'_i =$

Algorithm Insert (q);
begin
1. initialize: $i := 1$; $down_0 := \emptyset$; $h := \infty$
(* From invariant $\mathsf{INS}(i)$ (b), we know that $\widetilde{S}_i = S_i \cup down_{i-1} \cup \{q\}$. We want to determine $\widetilde{S}'_i$. Before that, we check if the data structure has to be rebuilt. *)
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**
$Build(\widetilde{S}_i)$; $h := i - 1$; goto **4.**
else
the old pivot p_i of S_i is also the pivot for $\widetilde{S}_i$
fi;
if $d(p_i, p) < \delta_i$ for some $p \in down_{i-1} \cup \{q\}$ **then**
$Build(\widetilde{S}_i)$; $h := i - 1$; goto **4.**
else
do nothing (* $d_i = d(p_i, S_i) = d(p_i, \widetilde{S}_i)$ *)
fi;
3. Determine $\widetilde{S}'_i$:
compute the set $down_i$ from its candidate set $D_i = S'_i \cup down_{i-1}$, see Eq. (6);
$\widetilde{S}'_i := D_i \setminus down_i$; $\widetilde{S}_{i+1} := S_{i+1} \cup down_i$; (* now $\widetilde{S}_{i+1} = (\widetilde{S}_i \setminus \widetilde{S}'_i) \setminus \{q\}$ *)
if $N_i(q, \widetilde{S}_i) = \emptyset$ **then**
add q to $\widetilde{S}'_i$; goto **4.** (* q is sparse in $\widetilde{S}_i$, and so $\widetilde{S}_{i+1} = S_{i+1}$ *)
fi; (* q is not sparse in $\widetilde{S}_i$ *)
add q to $\widetilde{S}_{i+1}$;
$i := i + 1$; goto **2.**
4. Update heaps:
(* **Invariant:** $q \notin \widetilde{S}'_\ell$ for $\ell < i$.
Also $\min\{i, h\}$ is $h = i - 1$, if a rebuilding was made.
Otherwise, $h = \infty$ and so $\min\{i, h\} = i$. *)
for $\ell := 1$ **to** $\min\{i, h\}$ **do**
forall $p \in down_\ell \setminus down_{\ell-1}$ **do**
removefromheap(p, ℓ)
od
forall $p \in down_{\ell-1} \setminus down_\ell$ **do**
addtoheap(p, ℓ)
od;
od;
if $q \in \widetilde{S}'_{\min\{i,h\}}$ **then**
addtoheap$(q, \min\{i, h\})$
fi;
end;

Figure 3: Algorithm Insert(q). The heap update procedures *addtoheap* and *removefromheap* called in step **4** will be given later.

$(S_{i+1} \cup S'_i) \cup down_{i-1} \cup \{q\} = S_i \cup down_{i-1} \cup \{q\}$ which equals $\widetilde{S}_i$ by $\mathsf{INS}(i)$ (b). Since $\widetilde{S}_{i+1}$ and $\widetilde{S}'_i$ are disjoint, it follows that $\widetilde{S}_{i+1} = \widetilde{S}_i \setminus \widetilde{S}'_i$.

Finally, if $q \notin \widetilde{S}'_i$, $\mathsf{INS}(i+1)$ (b) holds because, $\widetilde{S}_{i+1} = S_{i+1} \cup down_i \cup \{q\}$ by lines (16) and (20). ∎

Corollary 2 *At the end of step* **3** *of algorithm Insert, we have computed a uniform sparse partition for* $\widetilde{S}$ *according to Definitions 4 and 2.*

Proof: Refer to Figure 3. Let $\hat{\imath}$ denote the value of i after the last completion of step **3**. This particularly means that for each level $1 \leq j < \hat{\imath}$, no rebuilding has been made and $q \notin \widetilde{S}'_j$. By induction on the number of levels, $\mathsf{INS}(\hat{\imath})$ (a)-(b) hold. (We have already seen that $\mathsf{INS}(1)$ vacuously holds, forming the base of the induction. The induction step is established by Lemma 14.)

Invariant $\mathsf{INS}(\hat{\imath})$ (a) implies that the 5-tuples at levels $1, \ldots, \hat{\imath} - 1$ satisfy Definitions 4 and 2. Now, the last iteration at level $\hat{\imath}$ is either a rebuilding or produces a 5-tuple such that $q \in \widetilde{S}'_{\hat{\imath}}$.

In the former case, algorithm $Build(\widetilde{S}_{\hat{\imath}})$ computes a uniform sparse partition for $\widetilde{S}_{\hat{\imath}}$, and the result is a uniform sparse partition for the set $\widetilde{S}$.

In the latter case, another application of Lemma 14 establishes $\mathsf{INS}(\hat{\imath}+1)$ (a). Let $\widetilde{L}$ denote the number of levels of the partition at the end of step **3**. If $\widetilde{L} = \hat{\imath}$, then all the levels have been reconstructed and satisfy Definitions 4 and 2. Otherwise, if $\widetilde{L} > \hat{\imath}$, then some levels of the partition have not been reconstructed and thus $\widetilde{L} = L$. In this case, the 5-tuples for $\widetilde{S}_j$ are the old 5-tuples for S_j, $\hat{\imath} < j \leq L$, which fulfill the desired property anyway. Therefore all the 5-tuples $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{\delta}_i)$, $1 \leq i \leq \widetilde{L}$, are uniform and $\widetilde{S}_{i+1} = \widetilde{S}_i \setminus \widetilde{S}'_i$ for $1 \leq i < \widetilde{L}$. It follows that this set of 5-tuples is a uniform sparse partition for $\widetilde{S}$. ∎

Having established the correctness of the algorithm, we now go into the implementation details in order to establish the running time. First, as already promised, we examine the sizes of the *down* sets. The crucial fact which enables us to estimate the total size of the *down* sets is that at any level of the partition, the new point q is the only one that can make a previously sparse point non-sparse. We express this in the following lemma.

Lemma 15 *Let the sets* $down_j, 1 \leq j \leq i$, *be defined, and let* $p \in down_j$ *for a level* $j \in \{1, \ldots, i\}$. *Then*

(1) $p \in N_j(q)$ *and*

(2) $N_j(p, S) = \emptyset$.

Proof: The first claim is obvious from the definition of $down_j$, cf. Equation (6). The second claim is true because $p \in down_j$ implies $p \in S \setminus S_{j+1}$, and by Lemma 10, $N_j(p, S) = \emptyset$ for each $p \in S \setminus S_{j+1}$. ∎

Lemma 16 *Let the sets* $down_0, \ldots, down_i$ *be as defined in Equation (6). Then*

$$\left| \bigcup_{1 \leq j \leq i} down_j \right| \leq 3^D.$$

Proof: Assume that $p \in down_j$ for some $j \leq i$. Then $p \in N_j(q)$ and $N_j(p, S) = \emptyset$ by Lemma 15. Moreover, let $\Psi \in \{-1, 0, 1\}^D$ be such that $p \in b_j^\Psi(q)$. The partial neighborhood $N_j^\Psi(q)$ is the intersection of q's neighborhood with the neighborhood of p in the grid $\mathcal{G}_j$. Refer to Figures 1 and 2. Since $N_j(p, S) = \emptyset$, $N_j^\Psi(q)$ contains no point of $S \setminus \{p\}$.

Now, consider a point $p' \in down_\ell$ for any $\ell > j$. From Lemma 15, we know that $p' \in N_\ell(q)$. Furthermore, assume that p' is in the box of q's neighborhood with signature Ψ, i.e. $p' \in b_\ell^\Psi(q)$. Since $\delta_\ell \leq \delta_{j+1} \leq \delta_j/2$ by Lemma 1 (2), Lemma 9 (property (N.5)) gives $p' \in N_j^\Psi(q)$, from which it follows that p' must be identical to p.

This means that at levels $j+1 \leq \ell \leq i$, there cannot be any point in $down_\ell$ with signature Ψ except p itself. (Note that a point can be in several *down* sets.) It follows that for each $\Psi \in \{-1, 0, 1\}^D$, the set of points p in S such that there exists a $j \in \{1, \ldots, i\}$ satisfying $p \in down_j \wedge p \in b_j^\Psi(q)$ contains at most one element. ∎

Computing the *down* sets in constant time

We just proved 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 $\mathcal{G}_i$, we can compute $down_i$ in constant time. According to Equation (6), we want to find all $p \in S'_i \cup down_{i-1}$ such that $p \in N_i(q, S'_i \cup down_{i-1})$. How do we find these points? 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 and then search in the neighborhood of q. This proves that we can find $down_i$ in constant time.

Performing the changes in the data structures storing the sparse partition

Of course, the changes from the 5-tuple $(S_i, S'_i, p_i, q_i, \delta_i)$ to the 5-tuple $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{\delta}_i)$ also have to be performed in the data structures that actually store the 5-tuple. We will now fill in these details.

The operations that we have to care for are computing the new sparse set $\widetilde{S}'_i$ in line (16) and (18), and computing the new set $\widetilde{S}_{i+1}$ in lines (16) and (20) of algorithm Insert.

To compute $\widetilde{S}'_i = (S'_i \cup down_{i-1}) \setminus down_i$, we just have to insert resp. delete a constant number of points in the data structure storing the sparse set S'_i. To insert a point p into $\widetilde{S}'_i$ resp. $\widetilde{S}_{i+1}$, we add p to the list $\mathcal{L}'(b)$ resp. to the list $\mathcal{L}(b)$ where b is the box containing p. We also have to insert the box b into the box dictionary of the grid data structure, if it was not there before. This takes $O(1)$ expected time, see the discussion of the grid data structure in Subsection 3.2. (The same holds for the deletion of a box from the box dictionary.)

Now let us turn to the deletion of points. Note that during the insertion algorithm, deletions are performed in the sparse sets S'_i, more specifically there may be points that are in S'_i but are not in $\widetilde{S}'_i$. We can easily delete those points because we know that the lists $\mathcal{L}'(b)$ can only contain a constant number of points: at most one point at the start of the operation by the sparseness property, plus the points in $down_{i-1}$ that might have been tentatively inserted into the list. We remark here that instead of actually deleting the points of $down_i$ from the data structure storing the sparse set, we only *mark* them as deleted. The reason for this is that in step **4**, when we update the heaps, we need to access both the old set S'_i and the new set $\widetilde{S}'_i$. The actual deletions will be performed after step **4** has been completed.

The lists $\mathcal{L}(b)$ for the non-sparse set S_{i+1} can contain more than a constant number of points. However, observe that no point is ever deleted from a non-sparse set S_{i+1} during the insertion algorithm if no rebuilding is made.

To sum up, performing the changes from the old 5-tuple $(S_i, S'_i, p_i, q_i, \delta_i)$ to the new 5-tuple $(\widetilde{S}_i, \widetilde{S}'_i, \widetilde{p}_i, \widetilde{q}_i, \widetilde{\delta}_i)$ of the sparse partition takes $O(1 + |down_{i-1}| + |down_i|)$ expected time.

Lemma 17 *Steps* **1-3** *of algorithm Insert*(q) *take expected time* $O(\log n)$.

Proof: Consider one iteration of the steps **2** and **3**. If no rebuilding is made, the running time of step **2** is constant. (Recall that we assume that we can obtain a random number of $O(\log n)$ bits in constant time.) By the discussion in the two paragraphs before the lemma, the expected running time of step **3** at level i is $O(1 + |down_{i-1}| + |down_i|) = O(1)$.

We now give a probabilistic analysis for the insertion time, taking rebuildings into account. We show that the expected running time over all iterations of steps **1-3** is $O(\log n)$. The expectation is taken both over the new random choices and over the expected state of the old data structure.

Let the initial set of tuples be $(S_i, S'_i, p_i, q_i, \delta_i)$, $1 \le i \le n$, 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 overall running time X satisfies $X \le \sum_{i=1}^{n} T_i$. For $1 \le i \le n$, we have the following: with probability at most $\min(1, c/|S_i|)$ for some constant c, a rebuilding happens at level i and therefore $T_i = O(|S_i|)$ expected in this case. Otherwise, $T_i = O(1)$ expected. These expected time bounds stem from the running times of the hashing algorithms that are used to rebuild or to update the structure, respectively. Since the random choices made by the hashing algorithms are independent of the coin flips made by our algorithms Insert and Build to choose the pivots, we can multiply the probabilities of the events with the expected running times that are valid for the event and obtain an expected update time $\mathrm{E}(T_i) = O(1)$, independently of the previous state of the data structure.

Moreover, for any $1 \leq N \leq n$, $\sum_{i=N+1}^{n} T_i$ is bounded by the running time of the procedure $Build(S_{N+1})$. Hence, the expected value of this summation is bounded by $c' \cdot |S_{N+1}|$ for some constant c'. Choosing $N = \lceil \log n \rceil$, we obtain

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

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

Remark: Note that not only the running time at each level of the sparse partition, but also the number of levels is a random variable, and its value can be as high as n. This means in particular that the running times of consecutive update operations are not independent. In Section 6, we shall give a variant of the data structure which guarantees that the number of levels in the partition is $O(\log n)$ in the worst case. In this case, the probabilistic analysis will only be concerned with the probabilities of rebuildings. ∎

Discussion of the heap updates

We are now ready to discuss step **4** of the insertion algorithm. Heap updates are necessary when points move to a different level due to the insertion of q.

Assume point p moves to a different level. Then heap updates are necessary (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 algorithm inserts the values associated with p into the heap structure.

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 $H_\ell, i \leq \ell \leq i + D$: From Lemma 3 (3), we know that $d_i^*(p) = \min\left(\delta_i,\, d(p, S'_{i-D} \cup \ldots \cup S'_i)\right)$. Thus, a point $p \in S'_i$ can be associated with a heap in two different ways: First, there is a value $d_i^*(p)$ in H_i. Furthermore, for each $i \leq \ell \leq i + D$, there may be points $r \in S'_\ell$ such that $d(r,p)$ gives rise to $d_\ell^*(r)$ in H_ℓ.

Recall that $\widetilde{L}$ denotes the last level of the sparse partition *after* the update. In our heap update procedures given below, we want to rearrange the heaps such that heap $H_j, 1 \leq j \leq \widetilde{L}$, contains the values

$$\left\{ d_j^*(p) = \min\left(\widetilde{\delta}_j, d(p, \widetilde{S}'_{j-D} \cup \ldots \cup \widetilde{S}'_j)\right) \; : \; p \in \widetilde{S}'_j \right\}.$$

At the moment, the heaps contain the restricted distances w.r.t. the old sparse partition, except for the levels that have been rebuilt. We therefore take care that we only rearrange heaps at levels that have not been rebuilt. In step **4**, a parameter h occurs. It denotes the last level that has not been rebuilt if such a rebuilding has taken place. Otherwise, $h = \infty$. The heap update procedures are shown in Figures 4 and 5.

We have given simple procedures that do not use all the properties of the problem, since this gives a simpler proof and we are mainly interested in asymptotic complexity here. For example, one could check if the restricted distance stored at a heap item $it(r)$ changes because of removing resp. introducing the distance $d(r,p)$. Since recomputing all restricted distances in a small area around p takes only constant time, this would only save a constant factor, however.

Lemma 18 *After step* **4** *of algorithm Insert(q), the heap H_i stores the set $\{d_i^*(p) \; : \; p \in \widetilde{S}'_i\}$, for all levels $1 \leq i \leq \widetilde{L}$. Also, the running time of the procedures* addtoheap *and* removefromheap *is $O(1)$ plus the time spent on the heap operations. The number of heap operations that are performed in step* **4** *is constant.*

Proof: First note that at the beginning of step **4**, the new sparse partition is computed, and since the elements of S'_i that are not in $\widetilde{S}'_i$ have only been marked deleted, we have both sparse sets at hand at each level.

proc removefromheap(p, h);
begin
(* p starts moving at level i, i.e. $p \in S'_i$, but $p \notin \widetilde{S}'_i$ *)
$\text{DELETE}(H_i, it(p))$;
for $\ell := i$ **to** $\min\{i + D, h\}$ **do**
forall $r \in S'_\ell \cap \widetilde{S}'_\ell$ such that $d(r, p) < \delta_\ell$ **do**
$\text{CHANGE\_KEY}(it(r), d^*_\ell(r))$
od
od;
end;

Figure 4: Procedure removefromheap(p, h).

proc addtoheap(p, h);
begin
(* p stops moving at level j, i.e. $p \notin S'_j$, but $p \in \widetilde{S}'_j$ *)
compute $d^*_j(p)$; let r be such that $d^*_j(p) = d(r, p)$ if it exists;
$it(p) :=$ new item; $it(p).key := d^*_j(p)$; $it(p).point := p$; $it(p).point2 := r$;
$\text{INSERT}(H_j, it(p))$;
for $\ell := j$ **to** $\min\{j + D, h\}$ **do**
forall $r \in S'_\ell \cap \widetilde{S}'_\ell$ such that $d(r, p) < \delta_\ell$ **do**
$\text{CHANGE\_KEY}(it(r), d^*_\ell(r))$
od
od;
end;

Figure 5: Procedure addtoheap(p, h).

Notation: For each level i, we call points that remain sparse, i.e. the points in $S'_i \cap \widetilde{S}'_i$, the *passive* points, and the points that cease or start being sparse, i.e. the points in $(S'_i \setminus \widetilde{S}'_i) \cup (\widetilde{S}'_i \setminus S'_i)$, the *active* points.

Claim: Exactly the restricted distances that can change due to the change of the sparse partition and that have not been handled by rebuilding have been treated by the procedures shown in Figures 4 and 5.

Proof of Claim: At the beginning of step **4**, we know h, the index of the last level for which heap H_h has to be reconstructed, if a rebuilding has taken place. In this case, the data structure has been rebuilt at level $h+1$. (Otherwise $h = \infty$.) We can therefore guarantee that our heap update procedures do not treat levels whose heaps have already been correctly computed by a rebuilding.

Now consider the levels where the heaps have to be rearranged. Heap H_i contains the restricted distances of points $p \in S'_i$ to points in $S'_{i-D} \cup \ldots \cup S'_i$. For the active points, the claim is clear: These are the points in the symmetric difference of S'_i and $\widetilde{S}'_i$. By Fact 13, these are exactly the points in the symmetric difference of $down_{i-1}$ and $down_i$. Our heap update procedures are called exactly for these points and the restricted distances of these points are deleted in line (4) of removefromheap and inserted in line (6) of addtoheap, respectively.

For a passive point p, we only have to examine, at levels $j = i, \ldots, i-D$, the points that are

1. active at level j and
2. closer to p than the threshold distance δ_i.

These are exactly the points that are treated in lines (5)-(9) of removefromheap and in lines (7)-(11) of addtoheap. ∎ (Claim)

Also note that for every point that is treated by the heap update procedures, either the corresponding heap item is deleted, if it belongs to the set of points that ceases being sparse at that level, or its restricted distance according to the new sparse partition is computed (and inserted if it is a point that starts being sparse). This establishes the correctness of the heap update procedures and step **4** of algorithm Insert.

Now let us look at the running time of the heap update procedures. Each restricted distance can be computed in $O(1)$ time by Lemma 11. Moreover, from the proof of Lemma 11 we know that the restricted distances are computed by searching the area of at most $4D$ boxes away from p in the grids that store the sparse sets $S'_{i+l}, 0 \le l \le D$. Outside this area, the restricted distance of a point r cannot be affected by removal or insertion of p. Since we assume that the dimension D is fixed, the total number of heap operations carried out by the procedure is constant, and the time spent by the procedure not counting the heap operations is also constant. ∎

Theorem 1 *Algorithm Insert(q) correctly maintains the data structure and takes expected time $O(\log n)$.*

Proof: From Lemma 14, steps **1-3** establish that $S \cup \{q\}$ is stored uniformly as a sparse partition. Also, from Lemma 18, the heaps are maintained correctly by step **4**. This proves the correctness of the algorithm.

As shown in Lemma 17, steps **1-3** have expected cost $O(\log n)$. Now consider step **4**. From Lemma 16 we know that the heap update procedures are only called for a constant number of points. Since, by Lemma 18, one procedure call only performs $O(1)$ heap operations and, apart from these operations, performs only $O(1)$ additional work, the total time for step **4** is $O(\log n)$. ∎

4.2 The deletion algorithm

Now we come to the algorithm that deletes a point q from the data structure. Let $\widetilde{S}$ denote $S \setminus \{q\}$. 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.

An insertion ends at the level where the new point q is sparse. Therefore, assuming that $q \in S'_\ell$, we have to delete q from S'_ℓ and also from all the sets S_i, $1 \le i \le \ell$.

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

Algorithm Delete (q);
begin
1. initialize: $i := 1$; $up_0 := \emptyset$; $h := \infty$
(* From invariant $\mathsf{DEL}(i)$ (b), we know that $\widetilde{S}_i = (S_i \setminus up_{i-1}) \setminus \{q\}$. *)
2. check for rebuild:
(* we do not need to flip a coin for a new pivot *)
if q or an element of up_{i-1} is the pivot p_i or the nearest neighbor of p_i **then**
$Build(\widetilde{S}_i)$; $h := i - 1$; goto **4.**
fi; (* $d_i = d(p_i, S_i) = d(p_i, \widetilde{S}_i)$ *)
3. Determine $\widetilde{S}'_i$:
compute $up_i = \left\{ p \in S_i \ : \ N_i(p, S_i) = \{q\} \right\}$;
$\widetilde{S}'_i := (S'_i \cup up_i) \setminus up_{i-1}$; $\widetilde{S}_{i+1} := S_{i+1} \setminus up_i$; (* now $\widetilde{S}_{i+1} \cup \widetilde{S}'_i = \widetilde{S}_i \cup \{q\}$ *)
if $q \in \widetilde{S}'_i$ **then**
delete q from $\widetilde{S}'_i$; goto **4.** (* q is sparse in $\widetilde{S}_i$, and so $\widetilde{S}_{i+1} = S_{i+1}$ *)
fi; (* q is not sparse in $\widetilde{S}_i$ *)
delete q from $\widetilde{S}_{i+1}$;
$i := i + 1$; goto **2.**
4. Update heaps:
Completely analogous to algorithm Insert. At levels $1 \leq \ell \leq \min\{i, h\}$, we execute the heap update procedures 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 contributes a heap value.
end;

Figure 6: Algorithm Delete(q).

Although it looks natural to implement a deletion starting at the level ℓ where q is sparse, and then walking up the levels, it is much easier to implement the deletion algorithm in a top-down fashion, completely analogous to the insertion algorithm.

As already mentioned, points may move up some levels due to the deletion of q, analogous to the downward movement of points during an insertion. In the insertion algorithm, we collected in $down_i$ the points that were sparse at some level $j \leq i$ but that were no longer sparse at level i due to the insertion of q. That is, cf. invariant $\mathsf{INS}(i)$ (b) of the insertion algorithm,

$$down_i = \{p \in S \setminus S_{i+1} \ : \ p \in \widetilde{S}_{i+1}\}. \tag{7}$$

Now, we want to collect in up_i the points that are non-sparse at level i but will be sparse there after a deletion. We define $up_0 := \emptyset$ and for $i \geq 1$, if no rebuilding has been performed at levels $1, \ldots, i-1$,

$$up_i := \{x \in S_{i+1} \setminus \{q\} \ : \ x \notin \widetilde{S}_{i+1}\}, \tag{8}$$

Note that the deleted point q is not counted in this set, analogously to the treatment of the new point q in the insertion algorithm.

The deletion algorithm starts at the top level and moves downward, as algorithm Insert. See Figure 6. Let $i \geq 1$ and assume the deletion algorithm attempts to construct the 5-tuple for $\widetilde{S}_i$ without having made a rebuilding yet. Analogously to invariant $\mathsf{INS}(i)$ in algorithm Insert, we have

Invariant $\mathsf{DEL}(i)$ **:**

(a) Identical to $\mathsf{INS}(i)$, saying that the new 5-tuples at the levels $1, \ldots, i-1$ satisfy Definitions 4 and 2.

(b) The sets $up_j, 0 \leq j < i$, have been computed and $\widetilde{S}_i = (S_i \setminus up_{i-1}) \setminus \{q\}$.

Note that at the start of the algorithm, $\mathsf{DEL}(1)$ holds because $up_0 = \emptyset$.

To construct the 5-tuple for $\widetilde{S}_i$, the deletion algorithm first checks if a rebuilding has to be made, as does the insertion algorithm. Having done that, it constructs the new sparse set $\widetilde{S}'_i$ and, along with it, the non-sparse set $\widetilde{S}_{i+1}$. $\widetilde{S}'_i$ is computed from the previous sparse set S'_i by adding the points of up_i and deleting the points of up_{i-1}. Also, we obtain $\widetilde{S}_{i+1}$ by deleting the points of up_i from S_{i+1}. Now, q is still in $\widetilde{S}'_i$ or $\widetilde{S}_{i+1}$, depending on whether it was in S'_i or S_{i+1} before, respectively. Deleting q from the set containing it finishes the computation of $\widetilde{S}'_i$ and $\widetilde{S}_{i+1}$. If q was sparse at level i, then $S_{i+1} = \widetilde{S}_{i+1}$ and the construction of the new sparse partition is complete. (Note that up_i must be empty in this case.) Otherwise, we go into the next iteration and construct the 5-tuple for $\widetilde{S}_{i+1}$.

When the new sparse partition is computed, the heaps have to be updated. Analogously to the insertion algorithm, (i) $p \in up_{i-1} \setminus up_i$ means p *starts moving* at level i, i.e. $p \in S'_i$ and $p \notin \widetilde{S}'_i$, (ii) $p \in up_i \setminus up_{i-1}$ means p *stops moving* at level i, i.e. $p \notin S'_i$ and $p \in \widetilde{S}'_i$, and (iii) $p \in up_{i-1} \cap up_i$ means that p *moves through* level i, i.e. $p \notin S'_i$ and $p \notin \widetilde{S}'_i$. As before, the points that start or stop moving are causing heap updates.

From the similarity of the Equations (7) and (8), the arguments used to derive the bound on the size of the *down* sets carry over to the *up* sets, and thus we obtain $|\bigcup_{1\leq i\leq L} up_i| \leq 3^D$, see Lemma 16.

The computation of the *up* sets is slightly different from the computation of the *down* sets. Note that the initial definition of the *down* sets in Section 4 was a little bit different to (7) because we wanted to obtain a procedure to compute the *down* sets directly from the definition. Since examining the whole set that could possibly contain points of $down_i$, namely $S \setminus S_{i+1} = S'_1 \cup \ldots \cup S'_i$, could not be done efficiently, we defined $down_i$ recursively in terms of a "candidate set" $S'_i \cup down_{i-1}$ containing the points in $S'_1 \cup \ldots \cup S'_i$ that are eligible for being in $down_i$. From this candidate set, we were able to compute $down_i$ efficiently.

In contrast to the insertion case, the points that are non-sparse at level i but will be sparse there after a deletion, are all contained in S_i. We can compute the set up_i in constant time as follows. From Equation (8), it follows that $p \in up_i$ if and only if $N_i(p, S_i) = \{q\}$. 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.

Theorem 2 *Algorithm 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. ∎

We summarize the results of this section in the following theorem:

Theorem 3 *There exists a data structure that stores a set S of n points in $\mathbb{R}^D$ such that the minimal distance $\delta(S)$ can be found in $O(1)$ time, and all point pairs attaining $\delta(S)$ can be reported in time proportional to their number. The expected size of the structure is $O(n)$, and we can maintain the data structure as S is modified by insertions or deletions of arbitrary points, in $O(\log n)$ expected time per update. The algorithms run on a RAM and uses randomization. The bounds are obtained under the assumption that we know a frame that contains all the points that are in the set S at any time.*

Until now, the box dictionary, which stores the non-empty grid boxes, was implemented using perfect hashing. We needed to know a frame containing all the points in advance in order to employ this method. Clearly, we can also store these indices 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.

As a consequence, query and update operations on the box dictionary now take $O(\log n)$ deterministic time for a structure of size n. The requirement to know a frame containing all points in advance is not needed any more.

To update the entire data structure, we make an expected number of $O(\log n)$ dictionary operations plus a constant number of heap operations. Now, each dictionary operation takes $O(\log n)$ time. Hence, the expected update time is increased to $O(\log^2 n)$.

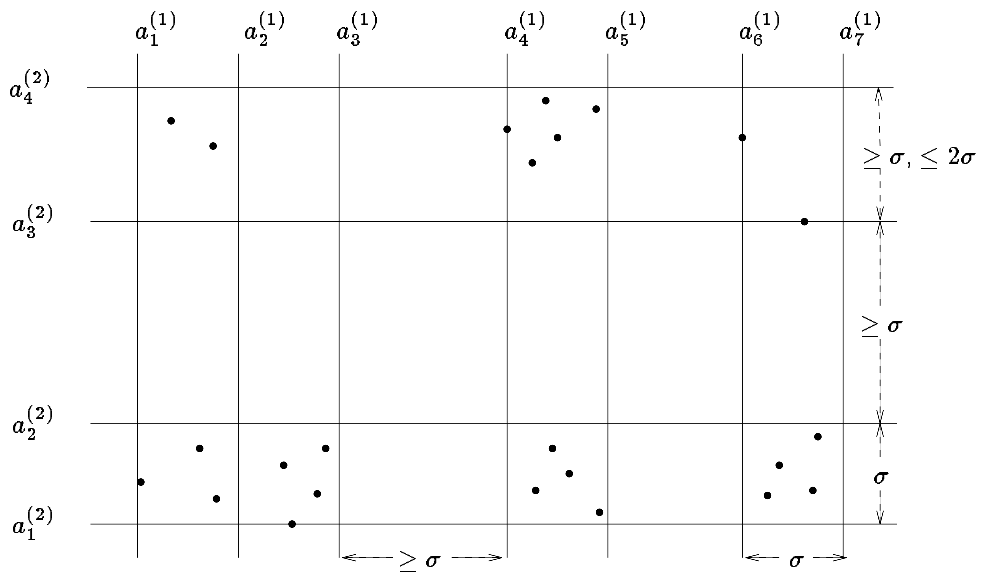

Figure 7: Example of a degraded δ-grid. It is dependent on the set stored in it.

Theorem 4 *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 $O(\log^2 n)$ expected time per insertion and deletion. The algorithms on this data structure use the floor function.*

5 An algebraic computation tree implementation

The solutions seen so far still use 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 computation 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 modify the algorithm of Theorem 4 by using a degraded grid for which we only need algebraic functions. The method we use already appears in [DLSS93] and [GRSS93]. We sketch the structure here and refer to these papers or [Sch93] for details.

Consider a standard grid of mesh size δ. Fixing the origin as a lattice point, we divide the space into slabs of width δ in each dimension. Since we can identify a slab using the floor function, this gives rise to an *implicit* storage of the slabs. To avoid the use of the floor function, we store these slabs *explictly*, by keeping a dictionary for the coordinates of its endpoints in each dimension.

In contrast to the standard grid, a degraded grid is defined in terms of the point set stored in it. To emphasize this, we use the notation $\mathcal{DG}_{\delta,V}$ for a degraded δ-grid defined by the points of a set $V \subseteq \mathbb{R}^D$ in comparison to the grid $\mathcal{G}_\delta$. In a degraded δ-grid $\mathcal{DG}_{\delta,V}$, all boxes have sides of length at least δ, and the boxes that contain a point of V have sides of length at most 2δ. See Figure 7.

The degraded grid can be maintained under insertions and deletions of points in logarithmic time. This increases the time bound for identifying the box containing a point from $O(1)$ to $O(\log n)$. However, this operation is always followed by a search in the box dictionary, which already takes logarithmic time in the tree implementation.

In order to implement our data structure, we only have to define the sparse sets S_i'. The alignment of boxes in slabs enables us to transfer the notion of neighborhood directly from standard grids to degraded grids. The neighborhood of a box consists of the box itself plus the $3^D - 1$ boxes bordering on it.

Consider a degraded δ-grid $\mathcal{DG}_{\delta,V}$. As in the grid case, the neighborhood of a point $p \in \mathbb{R}^D$ is defined as the neighborhood of the box $b_{\delta,V}(p)$ that contains p, i.e. $N_{\delta,V}(p) := N(b_{\delta,V}(p))$. The notion of partial neighborhood is also defined analogously. See Figure 1. We number the 3^D boxes in the neighborhood of a point p as described there, giving each box a *signature* $\Psi \in \{-1,0,1\}^D$. The box with signature Ψ is denoted by $b^{\Psi}_{\delta,V}(p)$. The boxes of p's neighborhood that are adjacent to $b^{\Psi}_{\delta,V}(p)$ form the *partial neighborhood of p with signature* Ψ, denoted by $N^{\Psi}_{\delta,V}(p)$.

We now define the neighborhood of a point relative to a set of points. For any set $\widehat{V}$, the neighborhood of p in $\mathcal{DG}_{\delta,V}$ relative to $\widehat{V}$, denoted by $N_{\delta,V}(p,\widehat{V})$, is defined as $N_{\delta,V}(p,\widehat{V}) := N_{\delta,V}(p) \cap (\widehat{V} \setminus \{p\})$. Note that in this definition, the set $\widehat{V}$ need not be identical to the *defining set* V of the degraded grid. As before, we say that a point p is *sparse* in the degraded grid relative to $\widehat{V}$ if $N_{\delta,V}(p,\widehat{V}) = \emptyset$.

The basis of correctness and running time of the grid algorithms were the neighborhood properties (N.1)-(N.5). We now adapt these to handle degraded grids. Two changes are needed. First, we change the constants in response to the fact that a non-empty box might now have side length up to 2δ. Second, although we defined the neighborhood relative to a set $\widehat{V}$ independently of the defining set V of the degraded δ-grid, a lot of properties will only continue to hold if $\widehat{V} \subseteq V$. The reason for this is the fact that boxes of a degraded δ-grid $\mathcal{DG}_{\delta,V}$ that do not contain a point of the defining set V may be unbounded. For example, this may cause a point $q \in \widehat{V}$ to be in the neighborhood $N_{\delta,V}(p)$ of a point $p \in \mathbb{R}^D$ even if it is arbitrarily far away from p.

Fact 19 *Let V be a set of points in $\mathbb{R}^D$, and let $p,q \in V$. Consider a degraded δ-grid $\mathcal{DG}_{\delta,V}$.*

(N.1') *If $q \notin N_{\delta,V}(p)$, then $d(p,q) > \delta$.*

(N.2') *If $q \in N_{\delta,V}(p)$, then $d(p,q) \leq 4D\delta$.*

(N.3') $q \in N_{\delta,V}(p) \Longleftrightarrow p \in N_{\delta,V}(q)$.

Lemma 20 *Let $0 < \delta' \leq \delta''/4$ be real numbers. Consider a degraded δ'-grid $\mathcal{DG}_{\delta',V'}$ and a degraded δ''-grid $\mathcal{DG}_{\delta'',V''}$, and let $p,q \in V'$. Then*

(N.4') $q \in N_{\delta',V'}(p) \Longrightarrow q \in N_{\delta'',V''}(p)$.

(N.5') *For any signature $\Psi \in \{-1,0,1\}^D$, let $q \in b^{\Psi}_{\delta',V'}(p)$. Then $q \in N^{\Psi}_{\delta'',V''}(p)$.*

Proof: Refer to the proof of (N.4) and (N.5). By the organization of the degraded grid boxes in slabs, the argument carries over directly, except that we have to care about the width of the slabs. Since $p,q \in V'$, the slabs containing p and q w.r.t. each coordinate have width at most $2\delta'$. Since $\delta' \leq \delta''/4$, equations (2) and (3) hold, which proves (N.4'). Once (N.4') is proved, stating that the neighborhood in the smaller grid is contained in the neighborhood of the larger grid, (N.5') follows completely analogous to (N.5) by equations (4) and (5), because these equations hold by the definition of the hyperplanes employed in the proof. See Figure 2. ∎

Now we are ready to define our degraded grid based sparse partition. Let $g_i := \delta_i/16D$. We store the set S_i in a degraded g_i-grid $\mathcal{DG}_{g_i,S_i}$. Analogously to Equation (1) for standard grids, we define

$$S_i' := \{p \in S_i : p \text{ sparse in } \mathcal{DG}_{g_i,S_i} \text{ relative to } S_i\}. \tag{9}$$

The sparse set S_i' will also be stored in a degraded g_i-grid $\mathcal{DG}_{g_i,S_i}$. Defining the sets S_i' for each i by Equation (9) yields a definition of a sparse partition analogous to the one given in Definition 4 for the grid case.

We adapt the abstract definition of the sparse partition (Definition 1) to degraded grids by changing "$\delta_i/2$" to "$\delta_i/4$" and "$\delta_i/4D$" to $\delta_i/16D$". The bounds in Lemma 1 then become $\delta_i \leq \delta_{i+1}/4$ and $\delta_L/16D \leq \delta(S) \leq \delta_L$, respectively. The constants in the other lemmas of Section 2 are changed analogously. Using a proof completely analogous to the one of Lemma 8, using (N.1')-(N.3') instead of (N.1)-(N.3), we get

Lemma 21 *Using the definition for S_i' given in Equation (9), we get a sparse partition according to Definition 1, with the constants changed as outlined above.*

The degraded grid based data structure:

For each $1 \leq i \leq L$:

- the pivot $p_i \in S_i$, its nearest neighbor q_i in S_i, and $\delta_i = d(p_i, q_i)$,
- S_i stored in a degraded g_i-grid $\mathcal{DG}_{g_i,S_i}$,
- S_i' stored in a degraded g_i-grid $\mathcal{DG}_{g_i,S_i}$,
- the heap H_i.

Now let us examine the update algorithms. In Section 4, we defined the sets $down_i$. The definition remains the same here, with the notion of neighborhood in degraded grids. The *down* sets describe the point movements between the levels of the sparse partition during an insertion. Similarly, the *up* sets contain the points that move to a different level during a deletion.

Due to the point movements between the levels, the defining set of the degraded grid at level i may contain extra points additional to the ones of S_i. We therefore use a distinguished name for the defining set of the degraded grid at level i, we call it V_i.

When the insertion algorithm reaches level i without having made a rebuilding before, it brings along the points of $down_{i-1}$ and the new point q, see Section 4. Therefore, $V_i = S_i \cup down_{i-1} \cup \{q\}$. It is important to see that, as for the sets S_i, we have

$$V_1 \supseteq V_2 \supseteq \ldots \supseteq V_L. \tag{10}$$

In the deletion algorithm, no additional point is introduced at any level, except in the sparse sets S_i'. We therefore have $V_i = S_i$. (Points that vanish from level i because they move upward may be deleted from the defining set V_i at the end of the deletion algorithm.)

Remark: The defining set V_i may differ from the non-sparse set at level i only *during an update algorithm.* After completion of an update operation, these sets are equal. Particularly, the update algorithms maintain the degraded grid $\mathcal{DG}_{\delta,S_i}$ for *both* S_i *and* S_i'. That is, a point p that is new in S_i due to an update is also added to the degraded δ-grid storing the sparse set S_i', even if it is not contained in S_i'. ∎

In the remainder of this section, we discuss the analysis of the insertion algorithm. The crucial point is the estimate on the size of the *down* sets. We transfer the relevant results to the degraded grid case, using the adapted neighborhood properties (N.4'), (N.5') given in Lemma 20. These properties hold with a restriction to the defining set of the degraded grid, whereas the original properties (N.4), (N.5) were valid without restriction to any point set. The nesting property (10) allows us to carry over the results nevertheless.

Analogously to the grid case, we use the following convention to describe neighborhoods in the sparse partition. For any point p, we let $N_i(p) := N_{g_i,V_i}(p)$. We use the analogous notation for the neighborhood relative to a set.

For the following statements, let $(S_i, S_i', p_i, q_i, \delta_i), 1 \leq i \leq L$, be a sparse partition as defined above, where V_i denotes the defining set of the degraded grid at level i.

The corresponding results to Corollary 1 and Lemma 10, obtained using (N.4'), are

- For any $1 \leq i < j \leq L$ and any $p \in V_j$, $\qquad N_j(p, V_j) \subseteq N_i(p, V_i)$.
- For any $p \in (S \setminus S_{i+1}) \cap V_i, 1 \leq i < L$, $\qquad N_i(p, S \cap V_i) = \emptyset$.

Now assume that algorithm Insert(q) processes the levels $1, \ldots, i$ without a rebuilding, and the sets $down_j, 1 \leq j \leq i$, are defined according to Equation (6). The corresponding statement to Lemma 15, which is obtained by using the above two statements, is

$$\text{Let } p \in down_j \text{ for a level } j \in \{1, \ldots, i\}. \text{ Then } p \in N_j(q) \text{ and } N_j(p, S \cap V_j) = \emptyset. \tag{11}$$

With these preparations, we can prove that Lemma 16 remains valid, i.e. $\left| \bigcup_{1 \leq j \leq i} down_j \right| \leq 3^D$.

We recall the proof of Lemma 16 together with the changes that are needed. The proof is now done with (11) and (N.5') replacing Lemma 15 and (N.5), respectively.

Assume that $p \in down_j$ for some $j \leq i$. Then $p \in N_j(q)$ and $N_j(p, S \cap V_j) = \emptyset$ by (11). Let $\Psi \in \{-1, 0, 1\}^D$ be a signature such that $p \in b_j^{\Psi}(q)$. Refer to Figures 1 and 2. Note that the boxes $b_j^{\Psi}(q)$ and $b_j(q)$ have side lengths between g_j and $2g_j$ in the degraded g_j-grid, because both p and q are in V_j. The partial neighborhood $N_j^{\Psi}(q)$ is equal to $N_j(q) \cap N_j(p)$. Since $N_j(p, S \cap V_j) = \emptyset$ by (11), $N_j^{\Psi}(q)$ contains no point of $(S \cap V_j) \setminus \{p\}$.

Now, consider a point $p' \in down_\ell$ for any $\ell > j$. Then $p', q \in V_\ell$, and we have $p' \in N_\ell(q)$ by (11). Assume that $p' \in b_\ell^{\Psi}(q)$. Since $\delta_\ell \leq \delta_{j+1} \leq \delta_j/4$, (N.5') gives $p' \in N_j^{\Psi}(q)$. We also have $p' \in V_j$, because $p' \in V_\ell$ and $V_\ell \subseteq V_j$ by the nesting property (10). However, we know from above that $N_j^{\Psi}(q)$ contains no point of $S \cap V_j$ except p. Therefore, $p' = p$.

This shows that, for each signature $\Psi \in \{-1, 0, 1\}^D$, all boxes $b_j^{\Psi}(q), 1 \leq j \leq i$, together contribute at most one element to the union $\bigcup_{1 \leq j \leq i} down_j$, which completes the proof. ∎

Now let us turn to the running time of the algorithm. Note that, when using a degraded grid data structure, the only difference in performance to the tree based grid structure employed at the end of Section 4 is that identifying the box containing a given point now takes $O(\log n)$ time on a structure of size n. This is irrelevant here since searching for that box in the box dictionary takes $O(\log n)$ time for both structures. The running time of the algorithm is therefore the same as in Theorem 4.

Theorem 5 *Let S be a set of n points in $\mathbb{R}^D$. 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.*

Note that the degraded grid not only depends on g_i, as in the grid case, but also on the set S_i stored in it (resp. on the set $V_i \supset S_i$ during the insertion algorithm). Actually, it even depends on the way S_i has developed by updates. This means that this data structure does no longer have the property that its distribution is independent of the history of updates. This does not affect the analysis of the algorithms, however.

6 Extensions

In this section, we give a variant of the data structure that achieves linear space in the worst case. The update time bounds on this structure are amortized, i.e. we have a bound on the expected running time of the whole update sequence. We also show that the algorithm executes an update sequence quickly with high probability.

We use the terms "hashing based implementation" and "tree based implementation" for the data structures of Theorem 3 and of Theorems 4 and 5, respectively.

6.1 A data structure with linear space in the worst case

Recall that the source of the expected space bound was the sum of the sizes of the non-sparse sets $S_1, \ldots, S_L$ of the sparse partition. To turn this into a worst case bound, we have to look at the algorithm *Sparse_Partition* given in Section 2. The data structures that we used to implement the sparse partition remain the same.

The idea to construct a sparse partition of linear size is simple. Refer to the description of the algorithm *Sparse_Partition* in Section 2. After picking the pivot randomly, we determine the set of sparse points induced by this random choice. If at least half of the points are sparse, we call the pivot *good* and take it. Otherwise, we discard it and make a new random choice, continuing this process until a good pivot is found. Note that this way of pivot selection resembles the iteration to find what was called a good hash function in the static hashing algorithm of [FKS84].

We make the following observations:

1. At least half of the elements of the set are good pivots, and so at most two trials are needed on average until a good pivot is found.

2. We can make the algorithm terminating in all cases by removing rejected pivot candidates from the set of eligible points. We can, on the other hand, analyze this as if we would consider all elements in each iteration, thereby only overestimating the cost.

3. The uniformness property (Definition 2) is lost. However, since at least half of the elements are good pivots, the probability of an element being the pivot right after the construction of a new sparse partition is at most $2/n$ for a set of size n.

4. Updates can gradually unbalance the data structure in the sense that more than half of the elements can become non-sparse. Unlike before, where this situation was controlled completely by the probabilistic analysis, we now enforce the essence of the balance condition "by hand" to ensure that the data structure uses linear space. That is, we count the number of update operations that affect a level of the data structure and rebuild after this count has reached a suitable constant fraction of the cardinality of the set at that level at the time of the last rebuilding.

We now make this precise. For convenience, we repeat the algorithm *Sparse_Partition* with the above described modification. We keep two global variables $last_i$ and $count_i$ to keep track of the sizes of the sets S_i of the partition during the update algorithms.

***last**$_i$* denotes $|S_i|$ after the last rebuilding that affected S_i, i.e. a rebuilding called at a level $j \leq i$.

***count**$_i$* denotes the number of update operations since the last rebuilding that affected level i.

Algorithm *Sparse_Partition*(S):

(i) $S_1 := S$; $i := 1$.

(ii) $V_i := S_i$; $last_i := |S_i|$; $count_i := 0$;

(iii) Pick $p_i \in V_i$ at random. Calculate $\delta_i = d(p_i, S_i)$.

(iv) Compute S_i' using one of the definitions of sparseness discussed before;

(v) If $|S_i'| < |S_i|/2$ then $V_i := V_i \setminus \{p_i\}$; goto (iii).

(vi) $S_{i+1} := S_i \setminus S_i'$;

(vii) If $S_{i+1} \neq \emptyset$ then set $i := i + 1$ and goto (ii).

Notation: An element $p \in S_i$ is *good* if, when picked in line (iii) of the above algorithm, $|S_i'| \geq |S_i|/2$ holds after line (iv).

Observation: At least half of the elements of S_i are good. (This can be seen by considering the sequence of nearest neighbor distances of the points in S_i that was employed in the proof of Lemma 2.) The elements of S_i that are not good are called *bad* elements.

Lemma 22 *For each i, let p_i be the pivot of the set S_i chosen by algorithm Sparse_Partition above. Then, for any point $p \in S_i$, $\Pr[p = p_i] \leq 2/|S_i|$.*

Proof: The probability to be the pivot is zero for a bad element. Otherwise, its probability is reciprocal to the number of good elements in the set. ∎

Lemma 23 *Let S be a set of n points in D-space. The modified algorithm Sparse_Partition produces a sparse partition of size $O(n)$. The hashing based implementation runs in $O(n)$ expected time, and the tree based implementation runs in $O(n \log n)$ expected time.*

Proof: The space bound is obvious, since the sizes of the sets S_i are geometrically decreasing. Each execution of the inner loop of the algorithm at level i runs in $O(|S_i|)$ expected time in the hashing based implementation and in $O(|S_i|\log|S_i|)$ deterministic time in the tree based implementation. Note that the logarithmic factor stems from the queries that are needed to find the sparse set S_i'. See the analysis of algorithm *Build*, which includes the construction of the sparse partition, in Lemma 12.

Since the expected number of executions of the inner loop is at most two at each level, the expected time at level i is $O(|S_i|)$ resp. $O(|S_i|\log|S_i|)$. Recall that the expected running time of the hashing algorithm is $O(|S_i|)$ independently of the coin flips made by algorithm *Sparse_Partition*.

The overall running time now follows, like the space bound, from the fact that the sizes of the sets S_i are geometrically decreasing. ∎

Note that the above lemma only concerns the building of the sparse partition itself, and not the complete algorithm *Build*, which also constructs the heaps for the sparse sets. This additional work can however be completed within the same bound. We need to distinguish between constructing the sparse partition itself on the one hand and the complete data structure on the other in the next subsection.

We can now turn to the update algorithms. First, we recall an assumption that we used for the main result in Section 4. In the update algorithms given there, the adversary has no clock to time the algorithm.

Let us remark here that allowing such an adversary does not destroy the update time bound. It only forces an amortized rebuild of the structure from time to time. If no rebuild occurs in an update operation, the adversary has revealed a constant number of points of the set not to be the pivot. These are the elements which are involved in the decision on whether a rebuilding is made or not, see step 2 of the update algorithms in Section 4. (Let the number of these elements be one for simplicity.) Then, after $n/2$ updates without rebuilding, the adversary would still not know which out of $n/2$ elements is the pivot. On the other hand, enough updates have taken place such that the data structure can be rebuilt and the costs are amortized in the update sequence.

Note that the probability of such an event is still high enough that we cannot make a purely probabilistic rebuilding. To see this, let $1/n$ be the probability of a rebuild in a set of size n. (We know that the actual probability is $O(1/n)$.) The probability that $n/2$ successive updates do not cause a rebuilding of the data structure is proportional to $(1-1/n)^{n/2} = \Theta(1)$. That is, the probability that no rebuilding occurs in $n/2$ updates is still constant, so the expected running time of the operation that performs the rebuilding which is needed to prevent the adversary from getting too much knowledge is at least linear.

Now let us return to the discussion of our modified data structure. We already mentioned that we need an amortized rebuilding anyway to retain the size of the sets of the sparse partition such that the total space used is linear. Therefore, taking the above discussion into account, we now allow a clocked adversary. This will not affect the running time.

We now discuss the insertion algorithm. We only sketch the modifications to the detailed description in Section 4. Refer to that section for notation. During the algorithm, c is a fixed constant. Step 2 (check for rebuild) is the only part that is changed.

Assume we insert the point q. We are at level i, $down_{i-1}$ has been computed, and $\widetilde{S}_i = S_i \cup down_{i-1} \cup \{q\}$. Now

1. $count_i := count_i + 1$; **if** $count_i \geq last_i/c$ **then** *Build*$(\widetilde{S}_i)$; stop;
2. **if** q or an element of $down_{i-1}$ is closer to the pivot p_i than its previous nearest neighbor
   then *Build*$(\widetilde{S}_i)$; stop;

The first item is the amortized rebuilding, and item 2 is the second part of the probabilistic rebuilding in the original algorithm. Note that the first part of that rebuild step—flipping a coin for a new pivot and rebuilding if the coin flip hits one of the elements in $down_{i-1} \cup \{q\}$—does not appear here. In the original algorithm, we needed this to ensure the uniformness of the pivots. Since we do not have a complete uniformness here anyway (the bad elements cannot be pivots), we can treat a newly inserted element as if it were a bad element.

The deletion algorithm is completely analogous to the description in Section 4. When the algorithm reaches level i, up_{i-1} has been computed and we have $\widetilde{S}_i = (S_i \setminus up_{i-1}) \setminus \{q\}$.

1. $count_i := count_i + 1$; **if** $count_i \geq last_i/c$ **then** *Build*($\widetilde{S}_i$); stop;

2. **if** q or an element of up_{i-1} is either the pivot p_i or its nearest neighbor q_i
   then *Build*($\widetilde{S}_i$); stop;

Here, the probabilistic rebuilding is exactly the same as in Section 4. The amortized rebuilding is analogous to that of the modified insertion algorithm.

It is clear that the above modifications do not affect the correctness of the update algorithms. The probabilistic rebuildings of the update algorithms ensure that the pivot condition is fulfilled. As long as no rebuilding is made, the algorithm uses a constant number of dictionary operations at each level. These cost $O(1)$ expected and $O(\log n)$ deterministic time in the hashing and the tree based implementation, respectively. The heap updates cost $O(\log n)$ time as before, for both implementations.

It remains to analyze the rebuildings. It is clear that the amortized rebuilding does not increase the running time, since rebuilding of a set of size m takes time $O(m)$ resp. $O(m \log m)$ and the number of updates until the next rebuilding occurs is $\Theta(m)$. Furthermore, by the estimate on the number of points that can move between the levels during an update operation, the balance condition that at least half of the elements are sparse at each level can be disturbed only by a constant amount of elements. Thus, with a suitable constant c, after m/c updates at a level of initial[1] size m, there are still $\Theta(m)$ sparse elements at that level. This ensures the property that (i) the data structure has $O(\log n)$ levels and (ii) the size of the structure is $O(n)$.

For the probabilistic rebuildings, we have to show that for each update that affects a level of size m, the probability of a rebuilding is c'/m for some fixed constant c'. It is clear that this is true directly after a rebuilding, with $c' = 2$. After that, the adversary gains knowledge about the possible pivots if no rebuilding is made. However, we have already seen that the adversary can exclude at most a constant number of points from being pivot in each update operation. Thus, similarly to the control over the sizes of the sets, after m/c updates at a level of initial size m, the adversary still has to consider $\Theta(m)$ possible pivots, which means that the probability of a rebuilding in item 2 of the above modifications is still $O(1/m)$. This means that the expected number of probabilistic rebuildings at a level of size m is only constant between two successive amortized rebuildings.

This proves the following theorem.

Theorem 6 *Let S be a set of n points in $\mathbb{R}^D$. There exists a randomized data structure for the dynamic closest pair problem that uses space $O(n)$ in the worst case and can maintained in $O(\log n)$ (for the hashing based implementation, under the assumption that we know a frame that contains all the points of S in advance) or $O(\log^2 n)$ (for the tree based implementation) expected amortized time per insertion and deletion.*

6.2 High probability bounds

The previous theorem yields the expected running time of the dynamic closest pair algorithm on a *sequence* of updates. We now discuss the reliability of the algorithm for such an update sequence.

By our modifications that lead to Theorem 6, the only random variable that has to be studied is the rebuilding cost. The rest of the update consists of $O(1)$ heap operations, which cost $O(\log n)$, and of $O(\log n)$ dictionary operations. (Recall that our data structure now has $O(\log n)$ levels in the worst case for a set of size n.) Hence, if no rebuilding occurs, we need $O(\log^2 n)$ worst case time in tree based implementation. In the hashing based implementation, we need $O(\log n)$ time with high probability, by using reliable dictionaries, see [DM90, DGMP92].

These hashing methods implement a dictionary with constant query and update time and linear building time, where the time bound for query is worst case and the time bounds for update and construction hold with high probability. More specifically, for a dictionary of size n, they hold with probability $O(1 - n^{-s})$ for any fixed integer s. We shall call such a probability *n-polynomial probability* in the following.

As long the set sizes in the sparse partition are $\Omega(n^\epsilon)$ for some $\epsilon > 0$, the bounds for reliable dictionaries stated above can be directly applied. In [Sch93], it is shown how to obtain an $O(\log n)$ time bound, with n-polynomial probability, to update the whole sparse partition in the case that no rebuilding occurs.

[1] i.e., after the last rebuilding

We now analyze the rebuilding cost. To make our task easier, let us assume that the data structure has exactly $\log n$ levels, of size $n/2^i, 0 \leq i < \log n$. (This can only—at least asymptotically—overestimate the cost.) For each level, we associate the running time of a rebuilding *called at that level* with it. Note that although this rebuilding also reconstructs the levels below, we charge the cost only to the calling level. We analyze each level separately. Recall that from time to time, we have to make an amortized rebuilding—not under probabilistic control—to enforce that the levels of the partition store sets of geometrically decreasing size. We call the time between two amortized rebuildings at a level a *phase*. We have seen before that the length of a phase, i.e. the number of updates, at a level storing a set of size m is m/c for some constant c. For simplicity, let us assume that a phase has length m. Then, for any data structure of size n, we analyze an update sequence that is divided into 2^i phases of length $n/2^i$ at level i, for $0 \leq i < \log n$.

Note that the amortized rebuildings that divide the phases are subsumed in the total cost of the operations. We therefore concentrate on the probabilistic rebuilding. We start by assuming a linear deterministic rebuilding cost and plug in the actual rebuilding cost afterwards. At a level of size m, we define the rebuilding cost to be m with probability $1/m$, and 0 with probability $1 - 1/m$. Recall that the actual rebuilding probability was $O(1/m)$, so the following analysis reflects the situation of our algorithm correctly. At level i, the rebuilding cost is then $n/2^i$, with probability $2^i/n$. Taking the 2^i phases at level i together, we make n updates, and with each update operation $\#j, 1 \leq j \leq n$, we associate a random variable

$$X_j^i := \begin{cases} n/2^i & \text{, with probability } 2^i/n \\ 0 & \text{, with probability } 1 - 2^i/n. \end{cases} \tag{12}$$

The variables $X_j^i, 0 \leq i < \log n, 1 \leq j \leq n$, are independent. Each variable bounds a rebuilding cost, up to a certain factor. So, even if there is a dependency between the actual rebuilding costs that give rise to the variables X_j^i, these costs are *independently bounded* by the above described values.

Fact 24 *Let $X_j^i, 0 \leq i < \log n, 1 \leq j \leq n$, be as defined above. Then*

$$X_j^i \leq n/2^i \quad \text{and} \quad \mathrm{E}\Big[\sum_{j=1}^{n} X_j^i\Big] = n.$$

The expectation $\mathrm{E}[\sum_{j=1}^n X_j^i]$, taken for all levels $0 \leq i < \log n$, describes the rebuilding cost which is relevant for the expected running time of a sequence of update operations, analyzed in Theorem 6. We are now heading for a tail estimate. The kind of estimate that we use is known under the name *Chernoff bounds*.

Lemma 25 ([HR90]) *Let $X_1, \ldots, X_n$ be independent random variables such that $0 \leq X_i \leq M$, $1 \leq i \leq n$. Let $\mathrm{E} = \mathrm{E}[\sum_{j=1}^n X_j]$. Then, for any $t \geq 1$,*

$$\Pr\Big[\sum_{j=1}^{n} X_j > t \cdot E\Big] \leq \left(\frac{e^{t-1}}{t^t}\right)^{E/M}.$$

Corollary 3 *Let the variables $X_1, \ldots, X_n$ be as in Lemma 25, and let $\mathrm{E} = \mathrm{E}[\sum_{j=1}^n X_j] = n$. Let s be a fixed integer. We examine the cases $M \leq n$ and $M \leq n/\log n$, choosing $t = \Theta(\log n/\log\log n)$ and $t = \Theta(1)$, respectively.*

1. *Let $M \leq n$. There exists a constant c' such that* $\Pr\Big[\sum_{j=1}^{n} X_j^i > c' \cdot n \log n/\log\log n\Big] = O(n^{-s})$.

2. *Let $M \leq n/\log n$. There exists a constant c' such that* $\Pr\Big[\sum_{j=1}^{n} X_j > c' \cdot n\Big] = O(n^{-s})$.

We now return to the analysis of the random variables $\sum_{j=1}^n X_j^i$, $0 \leq i < \log n$. By Fact 24, we have $M := \max X_j^i = n/2^i \leq n$ for any level $0 \leq i < \log n$ of the data structure.

We divide the levels into two ranges: the levels $0 \leq i < \log\log n$ are called the *coarse levels* and the levels $\log\log n \leq i < \log n$ are called the *fine levels*. For the analysis of the coarse levels, we use item 1 of the above corollary and get that, with n-polynomial probability,

$$\sum_{0 \leq i < \log\log n} \sum_{j=1}^{n} X_j^i = O(n \log n).$$

For the fine levels, note that $M := \max X_j^i = n/2^i \leq n/\log n$. We can therefore apply item 2 of Corollary 3 and get that, with n-polynomial probability,

$$\sum_{\log\log n \leq i < \log n} \sum_{j=1}^{n} X_j^i = O(n \log n).$$

Taking these estimates together, we obtain a rebuilding cost for a sequence of $\Theta(n)$ updates that is bounded by $O(n \log n)$ with n-polynomial probability, under the assumption that rebuilding takes linear deterministic time.

We now consider the actual rebuilding cost. Let us start with the hashing based implementation. By Lemma 23, the modified algorithm *Sparse_Partition* runs in $O(n)$ expected time on a set of size n. Particularly, for each i, $0 \leq i < \log n$, the expected running time to construct the 5-tuple at level i, for the subset $S_i \subseteq S$ of size at most $n/2^i$, is $O(n/2^i)$.

This is because, for each i, the number of executions of the inner loop, testing pivots until a good one is found, is bounded by the number of independent coin flips—with success probability 1/2—that are needed until a success occurs. This random variable, call it X, is geometrically distributed, and its expected value is two, which gives rise to the bound of Lemma 23.

The probability that X exceeds t, for any integer t, is 2^{-t}. Thus, the number of iterations of the pivot selection procedure is $O(\log n)$ with n-polynomial probability at each level $0 \leq i < \log n$ constructed by the algorithm.

In the hashing based implementation, the running time of one iteration of the pivot selection procedure (lines (iii)-(v) of the modified algorithm *Sparse_Partition*) is dominated by the time to build a grid data structure for the point set. Using a reliable dictionary, this can be done in time $O(n/2^i)$ at level i, with full n-polynomial probability. See [Sch93] for details.

Now, since the linear time bound for each execution of the inner loop as well as the logarithmic bound on the number of executions hold with n-polynomial probability, the running time of the whole algorithm, when called at level i for a set of size $n/2^i$, is $O(\sum_{j=0}^{i}(n/2^j)\ \log n) = O((n/2^i)\ \log n)$, with n-polynomial probability.

By now, we have analyzed the part of the rebuilding algorithm that constructs the sparse partition. The other part, namely computing the heaps for the sparse partition, can be done within the same time bound, in fact, in $O(n/2^i)$ deterministic time.

Now, by adding up the failure probabilities for all possible rebuildings that occur at level i during the update sequence, we infer that—also with n-polynomial probability—*all rebuildings* at this level have the claimed running time. Hence, we can use the previous high probability analysis, see Corollary 3 and below, only multiplying the bounds by $\log n$.

We obtain the following theorem.

Theorem 7 *Let S be a set of n points in $\mathbb{R}^D$. The hashing based implementation of the closest pair algorithm of Theorem 6 performs a sequence of $\Theta(n)$ updates on S, starting with a set of size n, in $O(n \log^2 n)$ time with n-polynomial probability, under the assumption that we know a frame that contains all the points that are in the set S at any time.*

Now let us turn to the tree based implementation. Since we have to do much more than just build a grid data structure the cost of the tree implementation, as described up to now, is a factor $\log n$ higher than for the hashing implementation. See Theorem 6. The expensive parts are query operations that cost $O(\log n)$ in the tree implementation compared to $O(1)$ if we use hashing.

There were two parts where we used queries in algorithm *Build*. First, in the construction of the sparse partition, we use queries to find the sparse points in a set. Second, when we build the heaps for the sparse partition computed before, we use queries to compute restricted distances. See Section 3. Actually, there is a way to get rid of the first part, by the following idea. During the building of the grid or degraded grid data structure, one can link each non-empty box with the non-empty boxes in its neighborhood, where we mean the occurrences of the boxes in the box dictionary (not only in the geometric representation, which is trivial). We can use these pointers to find the sparse points of the point set which is being stored *in linear time*, as follows: Walk through the list of non-empty boxes of the grid. With the help of the above described pointers, we can access the point lists associated with the neighboring boxes in constant time, replacing the queries in the box dictionary.

Note that we do not need to maintain these pointers dynamically under update operations. They are only used directly after the grid data structure has been built. We did not mention this point up to now because it cannot reduce the running time of the original algorithm *Build* of Section 3: the computation of restricted distances still blows up the running time by a logarithmic factor. For the grid algorithms given in the previous chapters, this feature was not needed either.

Now, however, let us return to the modified building algorithm that employs the computation of the sparse partition as described in the previous subsection. The algorithm that constructs the sparse partition repeatedly checks pivots until a good one is found. Recall that we do this $O(\log n)$ times in order to achieve n-polynomial reliability. However, computing the restricted distances has nothing to do with the construction of the sparse partition itself. We need to do this only for the final result that is returned by the algorithm *Sparse_Partition*. Hence, we now need $O(\log n)$ iterations of linear time work, plus one execution of an $O(n \log n)$ time procedure. It follows that the running time for algorithm *Build* on a set of size n is $O(n \log n)$ with n-polynomial probability in the tree implementation, as it was the case for the hashing implementation.

Using the analysis made before, we get that the total rebuilding time in the update sequence is $O(n \log^2 n)$ with high probability. This matches the running time of the other parts of the algorithm in the tree based implementation. We therefore have

Theorem 8 *Let S be a set of n points in $\mathbb{R}^D$. The tree based implementation of the closest pair algorithm of Theorem 6 performs a sequence of $\Theta(n)$ updates on S, starting with a set of size n, in $O(n \log^2 n)$ time with n-polynomial probability.*

7 Concluding remarks

In this paper, we have given the first solution to the fully-dynamic closest pair problem that achieves linear space and polylogarithmic update time simultaneously, leaving the question whether this goal can also be achieved by a deterministic algorithm. After a preliminary version of this paper was published, Kapoor and Smid [KS94] answered this question affirmatively with a method that has amortized update time $O(\log^{D-1} n \log\log n)$ for $D \geq 3$ and $O(\log^2 n/(\log\log n)^\ell)$ for the planar case $D = 2$, where ℓ is an arbitrary non-negative integer constant. It remains open whether the dynamic closest pair problem can be solved with $O(\log n)$ update time by a deterministic algorithm.

We have given several variants for our randomized data structure. Besides the variants that arise from the possible implementations of a grid data structure, we can implement the data structure such that it uses $O(n)$ expected space or $O(n)$ space in the worst case. For the latter variant, we gave high probability bounds for the running time of an update sequence. Note that we achieved the $O(n)$ worst case space solution at the cost of making the update time bounds amortized. It is open whether one can achieve the worst case space bound without sacrificing the bound on each single update operation.

Furthermore, note that the high probability bound on the update time is the same for the tree and the hashing implementation. Thus, this bound matches the expected time bound for trees, while there is an extra logarithmic factor for the hashing based implementation. The source of this slowdown is the rebuilding algorithm, the cost of which is crucial for the high probability running time of an update sequence.

Particularly, the main bottleneck that makes the rebuilding algorithm more expensive—in comparison to the variant that gives the expected time bound—is the pivot selection. Recall that, in order to find a good pivot, we needed only an expected constant number of tests, but a logarithmic number of tests was made to have this property with high probability. Each such test took linear time.

It is therefore interesting to find a way to make the pivot selection reliable without paying an extra logarithmic factor. This would improve the high probability running time of the hashing based implementation.

Finally, it is interesting to study applications of dynamic closest pair algorithms and to evaluate the practical performance of the techniques presented in this paper. These issues are addressed in a current project [Sch94].

References

[Ben83] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. 15th Annu. ACM Sympos. Theory Comput.*, pages 80–86, 1983.

[BS76] 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.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* McGraw-Hill, New York, NY, 1990.

[DD91] M. T. Dickerson and R. L. Drysdale. Enumerating k distances for n points in the plane. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 234–238, 1991.

[DE84] W. H. E. Day and H. Edelsbrunner. Efficient algorithms for agglomerative hierarchical clustering methods. *J. Classif.*, 1:7–24, 1984.

[DGMP92] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proc. 19th Internat. Colloq. Autom. Lang. Prog.*, volume 623 of *Lecture Notes in Computer Science*, pages 235–246. Springer-Verlag, 1992.

[DLSS93] 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.

[DM90] 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.

[FKS84] M. Fredman, F. Komlos, and E. Szemeredi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 17:538–544, 1984.

[GRSS93] M. J. Golin, R. Raman, C. Schwarz, and M. Smid. Simple randomized algorithms for closest pair problems. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 246–251, 1993.

[HR90] T. Hagerup and C. Rüb. A guided tour of Chernoff bounds. *Information Processing Letters*, 33, 1990.

[KM91] 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.

[KS94] S. Kapoor and M. Smid. New techniques for the dynamic closest pair problem. In *Proc. 10th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1994.

[LN89] R. J. Lipton and J. G. Naughton. Clocked adversaries for hashing. Technical Report CS-TR-203-89, Princeton Univ., 1989.

[Rab76] M. O. Rabin. Probabilistic algorithms. In J. F. Traub, editor, *Algorithms and Complexity*, pages 21–30. Academic Press, New York, NY, 1976.

[Sal92] J. S. Salowe. Enumerating interdistances in space. *Internat. J. Comput. Geom. Appl.*, 2:49–59, 1992.

[Sch93] C. Schwarz. *Data structures and algorithms for the dynamic closest pair problem.* PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 1993.

[Sch94] C. Schwarz. Dynamic closest pair algorithms: implementation and application. Unpublished manuscript, 1994.

[SH75] M. I. Shamos and D. Hoey. Closest-point problems. In *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 151–162, 1975.

[Smi91] M. Smid. Maintaining the minimal distance of a point set in less than linear time. *Algorithms Rev.*, 2:33–44, 1991.

[Smi92] M. Smid. Maintaining the minimal distance of a point set in polylogarithmic time. *Discrete Comput. Geom.*, 7:415–431, 1992.

[SSS94] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. *Algorithmica*, 12:18–29, 1994.

[Sup90] K. J. Supowit. New techniques for some dynamic closest-point and farthest-point problems. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 84–90, 1990.