RANDOMIZED DATA STRUCTURES FOR THE DYNAMIC CLOSEST-PAIR PROBLEM*

MORDECAI GOLIN†, RAJEEV RAMAN‡, CHRISTIAN SCHWARZ§, AND 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. 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. Even this version is more efficient than the best currently known deterministic algorithm for $D > 2$. Both results assume that the sequence of updates is not determined in any way by the random choices made by the algorithm.

Key words. computational geometry, proximity, dynamic data structures, randomization

AMS subject classifications. 68U05

1. Introduction. We consider the *dynamic closest-pair problem*: We are given an initially empty set S of points in D-dimensional space and want to keep track of the closest pair of points in S, as S is being modified by insertions and deletions of individual points. We assume that D is an arbitrary constant and that distances are measured in the L_t-metric for some fixed t, $1 \leq t \leq \infty$. Recall that 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

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

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, t will be implicit, and we will 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 [20] and Bentley and Shamos [2] gave $O(n \log n)$-time algorithms for solving the closest-pair problem in the plane and in arbitrary but fixed dimension, respectively. This running

*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 partially supported by NSF grant CCR-8918152 and by Hong Kong RGC/CRG grant 181/93E.

‡Algorithm Design Group, Department of Computer Science, King's College London, Strand, London WC2R 2LS, UK.

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

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

time is optimal in the algebraic computation tree model [1]. If we allow randomization as well as the use of the (non-algebraic) floor function, we find algorithms with better (expected) running times for the closest-pair problem. Rabin, in his seminal paper [16] on randomized algorithms, gave an algorithm for this problem which ran in $O(n)$ expected time [16, 5]. Since then, alternative methods with the same running time have been discovered. In addition to the randomized incremental algorithm presented in [11], there is a different approach, described by Khuller and Matias [12], 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 [23]. 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 [24] gave a data structure with $O(\log^D n)$ amortized update time which uses $O(n \log^{D-1} n)$ space. When both insertions and deletions are allowed, Smid [22] described a data structure which 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 [21], with improvements stemming from results of Salowe [17] and Dickerson and Drysdale [7], 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 [13] devised a deterministic data structure of linear size which 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 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. This time bound is obtained assuming that the floor function can be computed in constant time and that the algorithm has prior knowledge of a frame which contains all the points in S at any time (this assumption enables the algorithm to create and use dynamic hash tables). Without either of the above assumptions, we 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 [13] in the case $D = 2$. Indeed, our algorithm is the first to obtain polylogarithmic update time using linear space for the dynamic closest-pair problem.

Our results require that the updates are generated by an *oblivious* adversary, who fixes a worst-case sequence of operations in advance and reveals it to the data structure in an on-line manner. Hence, the adversary's knowledge of the internal state of the data structure (which is a random variable) is limited to that which can be obtained *a priori*, and in particular, the sequence of updates is not determined in any way by the random choices made by the algorithm.

Given a set S of points, the sparse partition 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 [15, 19].

The paper is organized as follows. In Section 2, we give an implementation-independent definition of 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. A grid-based implementation of the sparse partition is given in Section 3, and algorithms for updating the sparse partition are given in Section 4. In Section 5, we show how to modify the grid-based data structure in order to obtain an algorithm which fits in the algebraic computation tree model. Section 6 contains extensions of the method. We show how to modify the data structure to achieve $O(n)$ worst case space rather than 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. Section 7 contains some concluding remarks.

2. Sparse partitions. We start with some notation. Let S be a set of n points in D-dimensional space. 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 $d(p,S) := \min\{d(p,q) : q \in S \setminus \{p\}\}$. 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 us to find a single closest pair, but also to report all pairs of points attaining the minimal distance in time proportional to their number.

We now describe an idealized version of our data structure. Although the final data structure will be different in many significant ways, the idealized description may offer some insight into the working of the final data structure. For now, assume that the points are in general position, and define the *sparseness* of a point $p \in S$ as being simply $d(p,S)$. We will assume the existence of a data structure which, for some fixed $\delta > 0$, maintains a set T of points under insertions, deletions and queries of the form "is $d(p,T) > \delta$?" for an arbitrary point p, and "enumerate all $q \in T$ with $d(p,q) \leq \delta$". Insertions, deletions and the former query are assumed to take constant time, while the latter query takes constant time per element reported.

Our idealized data structure partitions the points of S based on their sparseness. Suppose the set S initially contains n points, and we pre-process S to obtain a distance threshold δ such that between $1/3$ and $2/3$ of the points have sparseness greater than δ. This pre-processing can be done in $O(n)$ expected time as follows: Simply pick a random $p \in S$, compute $\delta = d(p,S)$, and check how many points have sparseness greater than δ. The fundamental observation of [12] is that p's position in a list of the points in S ordered by sparseness is also random, and hence $O(1)$ such random trials suffice on average to compute a suitable threshold.

We now perform a sequence of cn updates to S, for some sufficiently small constant $c > 0$. Let S' denote the set of points in S at any time during this sequence of updates whose sparseness is greater than δ. Since the sparseness of a point changes only when its nearest neighbor in S changes, and a point can only be the nearest neighbor of $O(1)$ points in the set, each update can only change the sparseness of $O(1)$ other points. Hence, during this sequence of updates, S' will continue to contain a constant fraction of points in S. Furthermore, no point in S' will be part of a closest pair in S. At the end of this sequence of updates we can re-compute a suitable threshold in linear time, the cost of which can be amortized over the $\Theta(n)$ updates since the last threshold computation. Applying the partitioning idea recursively to the set $S \setminus S'$ then leads, modulo some details, to a solution to the closest-pair problem with $O(\log n)$ expected

amortized time per update—in essence, there will be $O(\log n)$ levels of recursion, and each update results in $O(1)$ amortized expected work per level of recursion.

The most significant problem with this idealized approach is that the "distance threshold" queries that we postulated appear difficult to answer in reality. However, we can quite easily answer the *approximate* versions of these queries, which can distinguish between the cases $d(p,T) > (1+\epsilon)\delta$ and $d(p,T) \leq \delta$ for some fixed $\epsilon > 0$, but may give an arbitrary answer for values in between (a grid-based implementation using the floor function will doubtless occur to a reader familiar with Rabin's algorithm [16]). This causes many problems with the above approach which we have not been able to overcome in a direct manner. The main differences are that the sets "sieved out" at each level have to be more carefully defined, and that the rebuilding rules now contain probabilistic components rather than being purely deterministic as above. A few more complications arise when we go from the grid-based algorithm to the algebraic one.

We now present an abstract framework which captures all the properties that we need from the sets sieved out at each level, and prove some basic facts about these sets in the abstract framework.

DEFINITION 2.1. *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 $d(p_i, S_i)/4D < d(p, S_i) \leq d(p_i, S_i)/2$, the decision may be arbitrary as far as the results of this section are concerned and will be made precise by the implementation later. Condition (c) is used while implementing updates to the sparse partition efficiently. Some readers may now wish to adjourn to Section 3 and read up to the end of the proof of Lemma 3.3, where a concrete implementation of a sparse partition is discussed, before continuing with the rest of this section.

LEMMA 2.2. *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, $\{S'_1, S'_2, \ldots, S'_L\}$ 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. Part (1) is obvious. To prove the first part of (2), let $1 \leq i < L$. Since $p_{i+1} \in S_{i+1}$, we know from Condition (c) in Definition 2.1 that there is a point $q \in S_i$

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 2.1 implies that $d(p, S_i) > \delta_i/4D$, and from the first part of (2), we conclude that $\delta(S) > \delta_i/4D \geq \delta_L/4D$. The inequality $\delta(S) \leq \delta_L$ obviously holds, because δ_L is a distance between two points of S. □

DEFINITION 2.3. *Let S_i be some set of points and let $(S_i, S'_i, p_i, q_i, \delta_i)$ be a 5-tuple chosen randomly from some distribution. This 5-tuple is called* uniform *if, for all $p \in S_i$, $\Pr[p = p_i] = 1/|S_i|$. Now let $(S_i, S'_i, p_i, q_i, \delta_i)$, $1 \leq i \leq L$, be a sparse partition for the set S chosen randomly from some distribution on all of the sparse partitions on $S_1 = S$. The set S is said to be* uniformly stored *by the sparse partition if all its 5-tuples are uniform.*

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 2.1.

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

LEMMA 2.4. *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 fulfils the requirements of Definitions 2.1 and 2.3. To prove the bound on the size of the structure, we first note that $L \leq n$ by Lemma 2.2. 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 2.1 implies that $r \in S'_i$, i.e., $r \notin S_{i+1}$.

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

We remark that the procedure *Sparse_Partition* is the essential component of Khuller and Matias' algorithm [12], where it was used as a filtering procedure to compute δ_L and only kept track of the current set S_i. Our dynamic algorithm stores the sets S_i and S'_i at all the levels. As we now show, the minimal distance is closely related to the sparse sets S'_i, i.e. the points that were thrown away in each step of the iteration in [12]. This enables us to use the sparse partition to find $\delta(S)$ quickly.

DEFINITION 2.5. *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 2.6. *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. The proof of (1) is obvious. To prove (2), let $q \in S'_j$, where $1 \leq j < i - D$. Since $d(p,q) > \delta_j/4D$, Lemma 2.2 implies that

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

(3) follows immediately from (2). □

LEMMA 2.7.

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

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

It remains to prove that we can restrict the value of i to $L-D, L-D+1, \ldots, L$. We know from Lemma 2.6 (1) that $\min_{p\in S'_i} d_i^*(p) > \delta_i/4D$. Moreover, we know from Lemma 2.2 (2), that for $i < L - D$, $\delta_i/4D \geq \delta_{L-D-1}/4D \geq (2^{D+1}/4D) \cdot \delta_L \geq \delta_L \geq \delta(S)$. □

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 2.7, we will augment the sparse partition with a data structure which 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 a *heap*. (See e.g.[6, Chapter 7].) A heap storing n items can be built in linear time, and the operations *insert*(item), *delete*(item), and *change_key*(item, key)—which changes the key of the item item to the value key—are supported in $O(\log n)$ time. Also, operation *find_min*(H) can be performed in $O(1)$ time, and *find_all_min*(H), which returns all A items with minimum key, can be performed in time $O(A)$.

So, for each $i \in \{1, \ldots, L\}$, we maintain a min-heap H_i which 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 following sections, where we also describe the exact contents of our heap items.

LEMMA 2.8. *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 2.7 characterizes $\delta(S)$ as a minimum of certain restricted distances. In particular, Lemma 2.7 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 2.1 and 2.3.
- The heaps H_1, H_2, ..., H_L, where H_i stores the set of restricted distances $d_i^*(p)$, cf. Definition 2.5, 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. A grid-based 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. $N_\delta(p) := N(b_\delta(p))$. 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 where V and δ are understood from the context we will simply say that p is *sparse*.

The following observations follow directly from the definitions:

LEMMA 3.1. *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* $d(p,q) \leq 2D\delta$
(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 $i \geq 1$, let

$$S_i' := \{p \in S_i \ : \ p \text{ sparse in } \mathcal{G}_{\delta_i/4D} \text{ relative to } S_i\}. \tag{3.1}$$

For convenience, we now modify the the abstract definition of the sparse partition given in Definition 2.1 and replace it by one which is defined in terms of grid neighborhoods. Recall that in Definition 2.1 S_i' was not fully specified; the definition only gave conditions, specifically (b.1), (b.2), (c) and (d), which S_i' had to satisfy. We now replace those conditions by Equation (3.1); with this new definition the sparse partitions are totally specified and we will be able to use them in Section 4 to develop and later analyze our update algorithms.

DEFINITION 3.2. *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:*

1. *For* $i = 1, \ldots, L$:
   (a) $\emptyset \neq S_i \subseteq S$;
   (b) $p_i, q_i \in S_i$ *and* $p_i \neq q_i$ *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* $1 \leq i \leq L-1$, $S_{i+1} = S_i \setminus S_i'$.

LEMMA 3.3. *Let* $(S_i, S_i', p_i, q_i, \delta_i)$, $1 \leq i \leq L$, *be a set of 5-tuples satisfying Definition 3.2. Then this set of 5-tuples also satisfies Definition 2.1.*

Proof. We only have to prove Conditions (b) and (c) of Definition 2.1. Let $1 \leq i \leq L$ and let $p \in S_i$. First assume that $p \notin S_i'$. Then, there is a point $q \in S_i$ which is in the neighborhood of p. By (N.2), $d(p, S_i) \leq d(p,q) \leq 2D \cdot \delta_i/4D = \delta_i/2$. 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 $1 \leq i < L$ 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 $d(p,q) \leq \delta_i/2$.

In Figure 3.1 we provide an example of a sparse partition based on Definition 3.2 □

We now come to some additional properties of the sparse partition as defined in Definition 3.2 that will be used in 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 now define the notion of *partial neighborhood* of a point p. (See Figure 3.2.) 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. In particular, $N_\delta^{0,\ldots,0}(p) = N_\delta(p)$, i.e. the partial neighborhood with signature $0, \ldots, 0$ is the whole neighborhood of p.

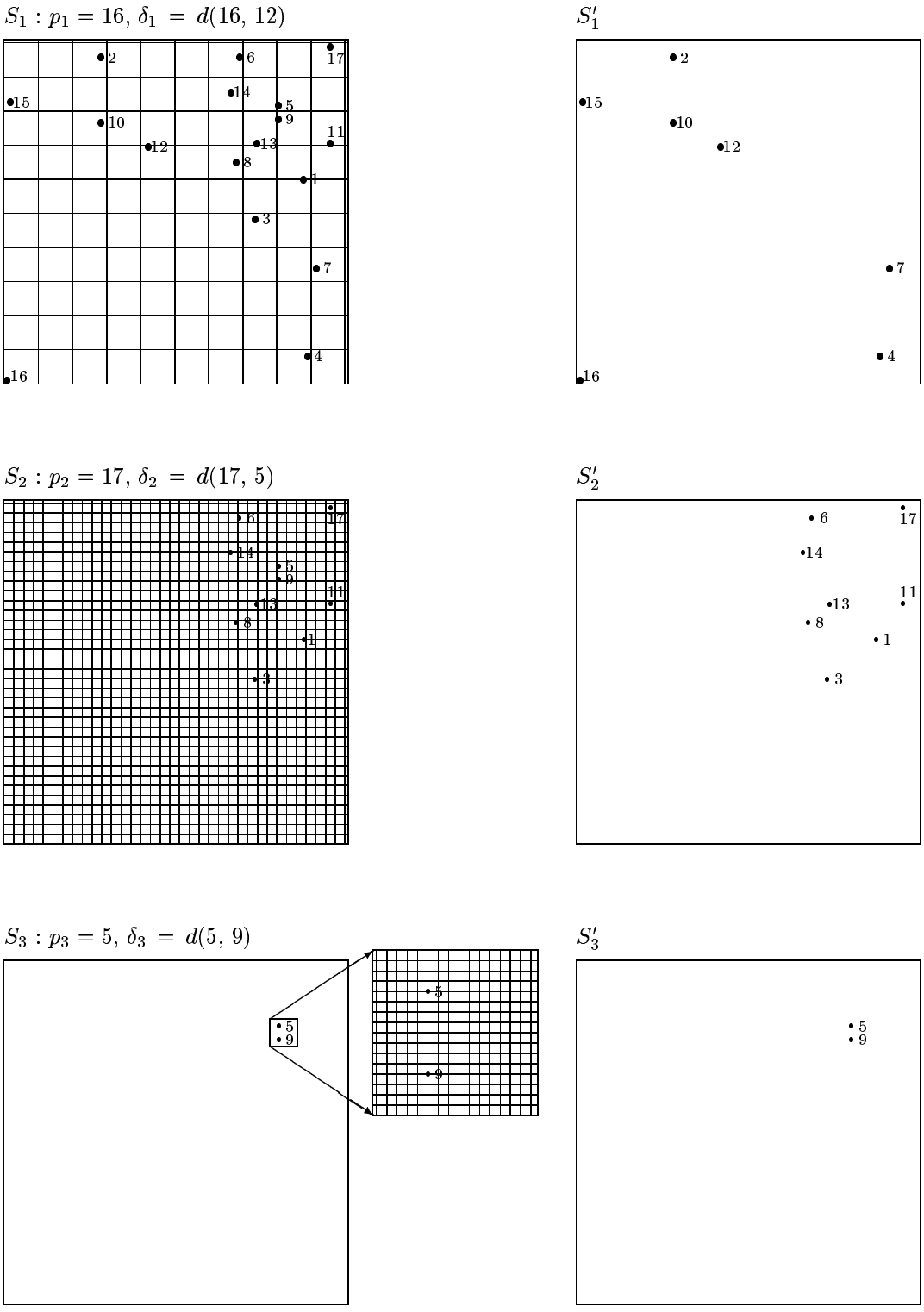

FIG. 3.1. *A sparse partition. Although the sets S_i' are also stored in grids, we have not shown the corresponding grids.*

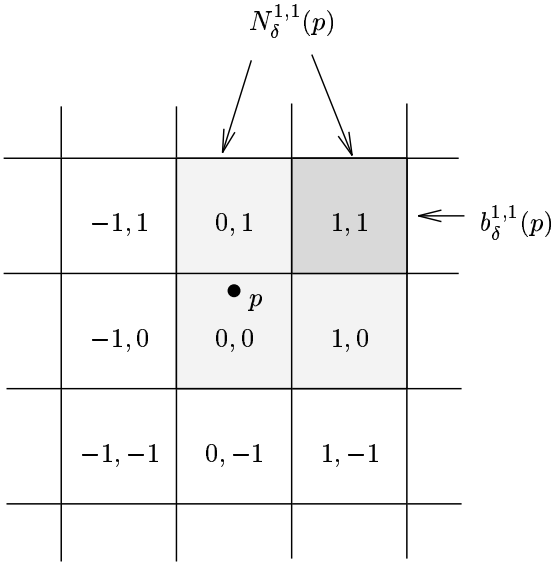

FIG. 3.2. *The neighborhood of a point p in $\mathcal{G}_\delta$. The dark shaded area denotes the box $b_\delta^{1,1}(p)$ in the upper right corner of p's neighborhood. This box also belongs to $N_\delta^{1,1}(p)$, the partial neighborhood of p with signature $1,1$. The light shaded area shows the other three boxes of $N_\delta^{1,1}(p)$.*

The following properties will let us relate the neighborhoods of a point in different grids and more specifically in the different grids that correspond to different levels of the same sparse partition.

LEMMA 3.4. *Let $0 < \delta' \leq \delta''/2$ be real numbers and let $p \in \mathbb{R}^D$. Then*
(N.4) $N_{\delta'}(p) \subseteq N_{\delta''}(p)$.
(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 3.3.

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 $1 \leq j \leq D$:

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

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

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

Figure 3.3 shows the neighborhoods of p in the two grids $\mathcal{G}_{\delta'}$ and $\mathcal{G}_{\delta''}$.

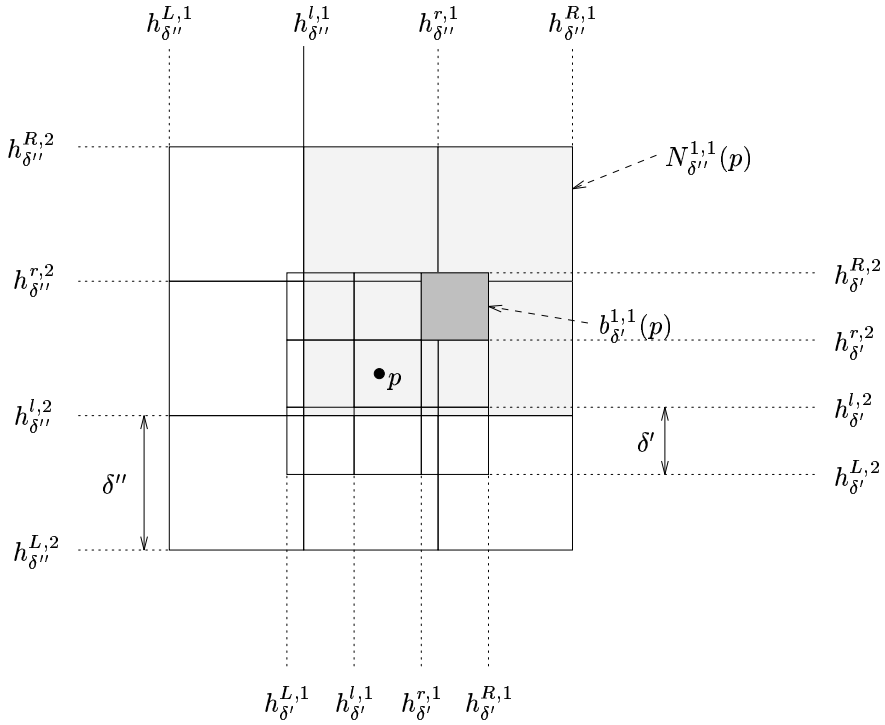

FIG. 3.3. *The neighborhoods of a point p in grids $\mathcal{G}_{\delta'}$ and $\mathcal{G}_{\delta''}$ where $\delta' \leq \delta''/2$.*

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

$$h^{L,j}_{\delta'} \geq h^{L,j}_{\delta''} \tag{3.2}$$

$$h^{R,j}_{\delta'} \leq h^{R,j}_{\delta''} \tag{3.3}$$

for $1 \leq j \leq D$. These facts are equivalent to $N_{\delta'}(p) \subseteq N_{\delta''}(p)$, which is claim (N.4).

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

$$h^{r,j}_{\delta'} \geq h^{l,j}_{\delta''} \tag{3.4}$$

$$h^{l,j}_{\delta'} \leq h^{r,j}_{\delta''} \tag{3.5}$$

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 which 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 3.2. 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 abbreviations $\mathcal{G}_i := \mathcal{G}_{\delta_i/4D}$ 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 3.5. *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 3.4 with $\delta'' = \delta_i/4D, \delta' = \delta_j/4D$, noting that $\delta' \leq \delta''/2$ by Lemma 2.2. □

In particular, 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 3.5 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 3.6. *Let $(S_i, S_i', p_i, q_i, \delta_i)$, $1 \leq i \leq L$, be a sparse partition. Then, for any $p \in S \setminus S_{i+1}, 1 \leq i < L$,*

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

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 3.5 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 have $p \in N_i(q)$, 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. We now explain how to store the point sets involved in the sparse partition of our input set S. Let δ be a grid size and $V \subseteq S$ a subset of S. We store each each non-empty box in the grid $\mathcal{G}_\delta$ in a hash table, using the index of the box in $\mathcal{G}_\delta$ as a key. Associated 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*.

The dynamic perfect hashing algorithm of [10] allows us to store a set of integer-valued keys in linear space such that the information stored at a given key can be accessed in $O(1)$ worst case time, and permits a key to be inserted or deleted in $O(1)$ expected time. This algorithm needs to know in advance the range of integers from which the keys to be inserted or deleted may be drawn, but this range can easily be computed from δ if we know a frame containing all the points in S.

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, and by doing this for each point in V we can, in linear time, find the subset $V' \subseteq V$ of sparse points in V.

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 $\mathcal{G}_i$ which is non-empty w.r.t. 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 $\mathcal{L}'(b)$ normally contains at most one point, by the sparseness property of the set S_i', but may temporarily contain more than one point during an update.

Now let us turn to the heaps. The key of an item in heap H_i is the value $d_i^*(p)$ for some $p \in S_i'$. Let q—if it exists—be such that $d_i^*(p) = d(p,q) < \delta_i$, and let l be such that $0 \le l \le D$ 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.

We are now in a position to describe a complete procedure which constructs the sparse partition and its auxiliary data structures. It will be convenient to have two slight variants of the procedure. The first, called $Build(T, j)$, takes as arguments a set of points T and an integer j, and stores set T uniformly in levels $j, j+1, \ldots$ of a sparse partition. The second, $Near\_Build(T, p, j)$, again stores the given set of points T uniformly in levels $j, j+1, \ldots$ of a sparse partition, but uses the given point $p \in T$ as the pivot for level j. In all invocations of $Near\_Build$, p will be chosen at random from T.

Algorithm $Near\_Build(T, p, j)$

1. $i := j$; $S_i := T$; $p_i := p$.
2. 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 $S_{i+1} := S_i \setminus S_i'$; choose a random point $p_{i+1} \in S_{i+1}$; set $i := i+1$; and goto 2.

Algorithm $Build(T, j)$

Choose a random point $p \in T$ and call $Near\_Build(T, p, j)$.

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 3.7. *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 2.6 (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 $\mathcal{G}_j$, $i - D \leq j \leq i$, in which the point q can possibly appear: 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 in which p is located. Finally, because of the sparseness of the sets S'_j, there can be at most one point in each grid box and 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 3.8. *Let T be a set of points in $\mathbb{R}^D$. The procedures $Build(T, j)$ and $Near\_Build(T, p, j)$ complete in $O(|T|)$ expected time and produce sparse partitions of $O(|T|)$ expected size.*

Proof. For $k = 0, 1, \ldots$, consider the k-th iteration of algorithm $Build(T, j)$, for which the loop index i has value $j+k$. 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'_l, $j \leq l \leq j + k$, available in the k-th iteration of the algorithm, we can apply Lemma 3.7 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 2.4 shows that this quantity is $O(|T|)$.

The analysis for $Near\_Build$ is similar. □

Recall that given this data structure, we can find the closest pair in S in $O(1)$ time by Lemma 2.8.

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. Then, we give the deletion algorithm.

4.1. The insertion algorithm. We start with an intuitive description of the insertion algorithm. Let S be the current set of points, and assume we want to insert the point q. Further assume that S is already uniformly stored in the sparse partition. Our goal is to randomly build a sparse partition which uniformly stores $S \cup \{q\}$.

If we were to build a uniform sparse partition of $S \cup \{q\}$ from scratch then q would be the pivot of S_1 with probability $1/(|S_1| + 1)$; otherwise one of the points in S is chosen at random as the pivot. By assumption, p_1 (the pivot of S_1) is a random element of $S_1 = S$. Therefore, 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 with probability $1/(|S_1|+1)$. If q is chosen, then we discard everything and run $Near\_Build(S_1 \cup \{q\}, q, 1)$, terminating the procedure. The latter event happens, however, only with probability $1/(|S_1| + 1)$ and so its 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 [8].) This means that δ_1 changes only if p_1 is one of these points. Since the updates are assumed to be independent of 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 $Build(S_1 \cup \{q\}, 1)$ 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 $2 \leq i \leq L$. 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. Two of the three cases that may occur while constructing the first 5-tuple are given in Figures 4.1 and 4.2. The algorithm for constructing the new 5-tuples for $\widetilde{S}_i, i > 1$ follows the same general idea. We now describe more formally how to construct the new 5-tuples for $\widetilde{S}_i, i \geq 1$, and extend the notion of the set $down_1$ from the first level to the other levels of the sparse partition.

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

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

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

(a.1) $q \in \widetilde{S}_j$ and the set of 5-tuples $(\widetilde{S}_j, \widetilde{S}'_j, \widetilde{p}_j, \widetilde{q}_j, \widetilde{\delta}_j)$, satisfies Definitions 3.2 (sparse partition) and 2.3 (uniformity), 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 $down_j, 0 \leq j < i$, 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. Thus, 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 take the new point q as new pivot with probability $1/(1+|S_1|)$. 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 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. This would result in a rebuilding (see Figure 4.3), but as the probabilities only increase by a constant factor, the effect is negligible.

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

$$down_i := N_i(q, D_i) = \{x \in D_i \,:\, x \in N_i(q)\}\,. \tag{4.1}$$

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 $\widetilde{S}_i$.)

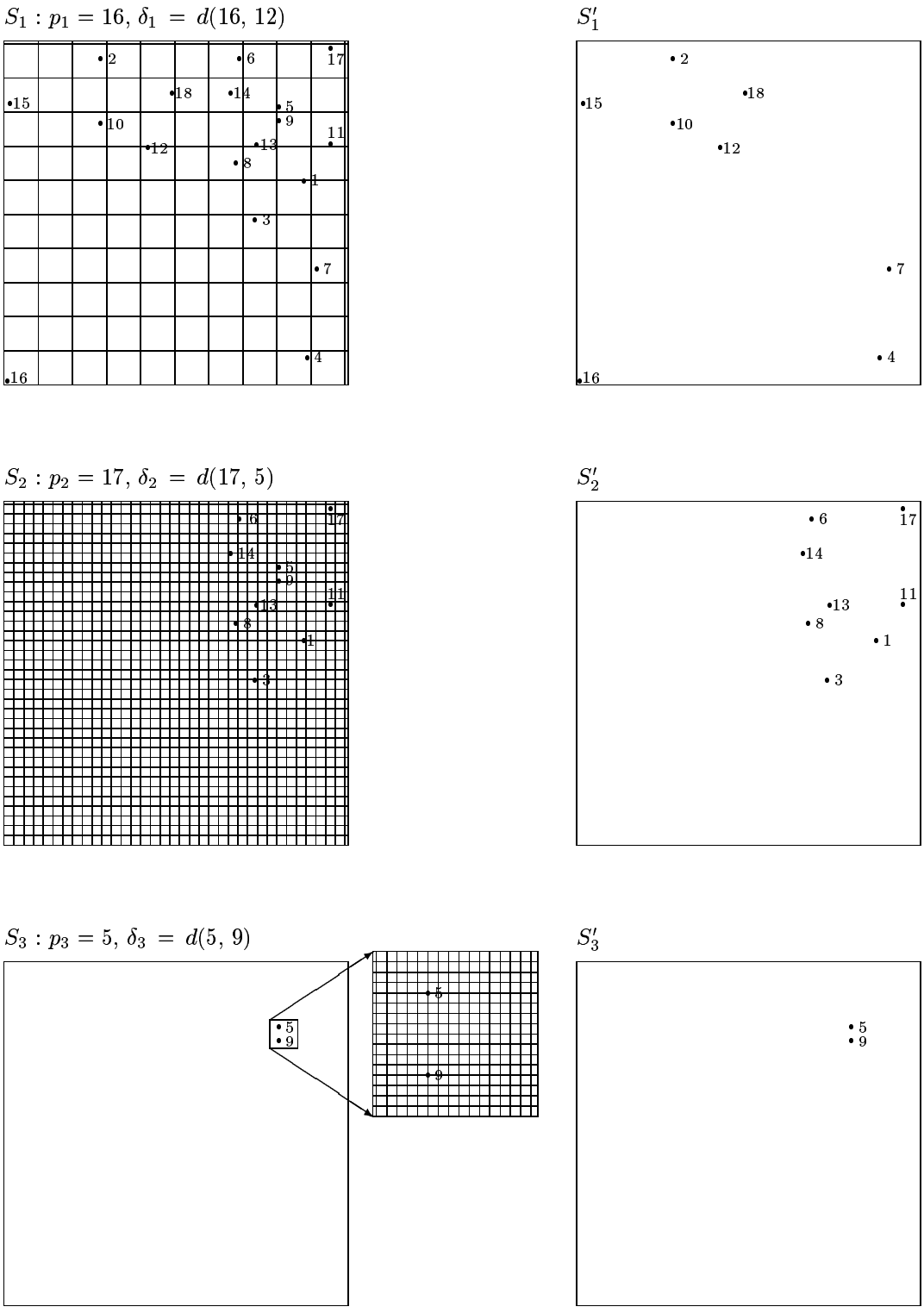

FIG. 4.1. *This is the sparse partition that resulted when point 18 was inserted into the sparse partition of the previous example. Point 18 was not chosen to be the pivot and was sparse in* S_1 *with the old pivot* $p_1 = 16$. *Thus* $18 \in S_1'$.

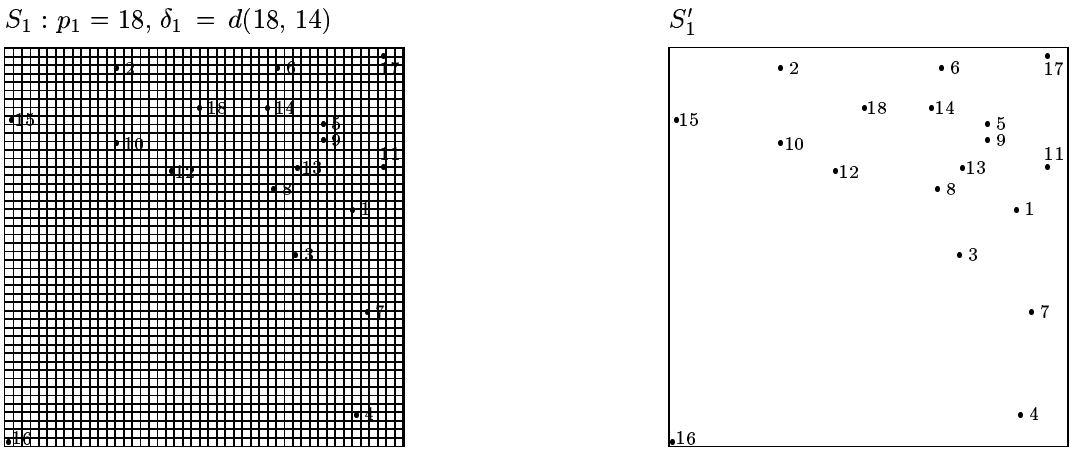

FIG. 4.2. *This is the sparse partition that results when the same point 18 is inserted as in the previous example but with point 18 now being chosen to be the pivot, something that occurs with probability* 1/18. *Note that in this case every point in* S_1 *becomes sparse so the partition only has one level.*

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 invariant $\mathsf{INS}(i)$ (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:

LEMMA 4.1. *Let* p *be a point in* S*:*

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

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

(iii) $p \in down_{i-1} \cap down_i \iff p \notin S'_i$ *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 4.4.

It remains to describe the procedures that actually perform the heap updates. Be-

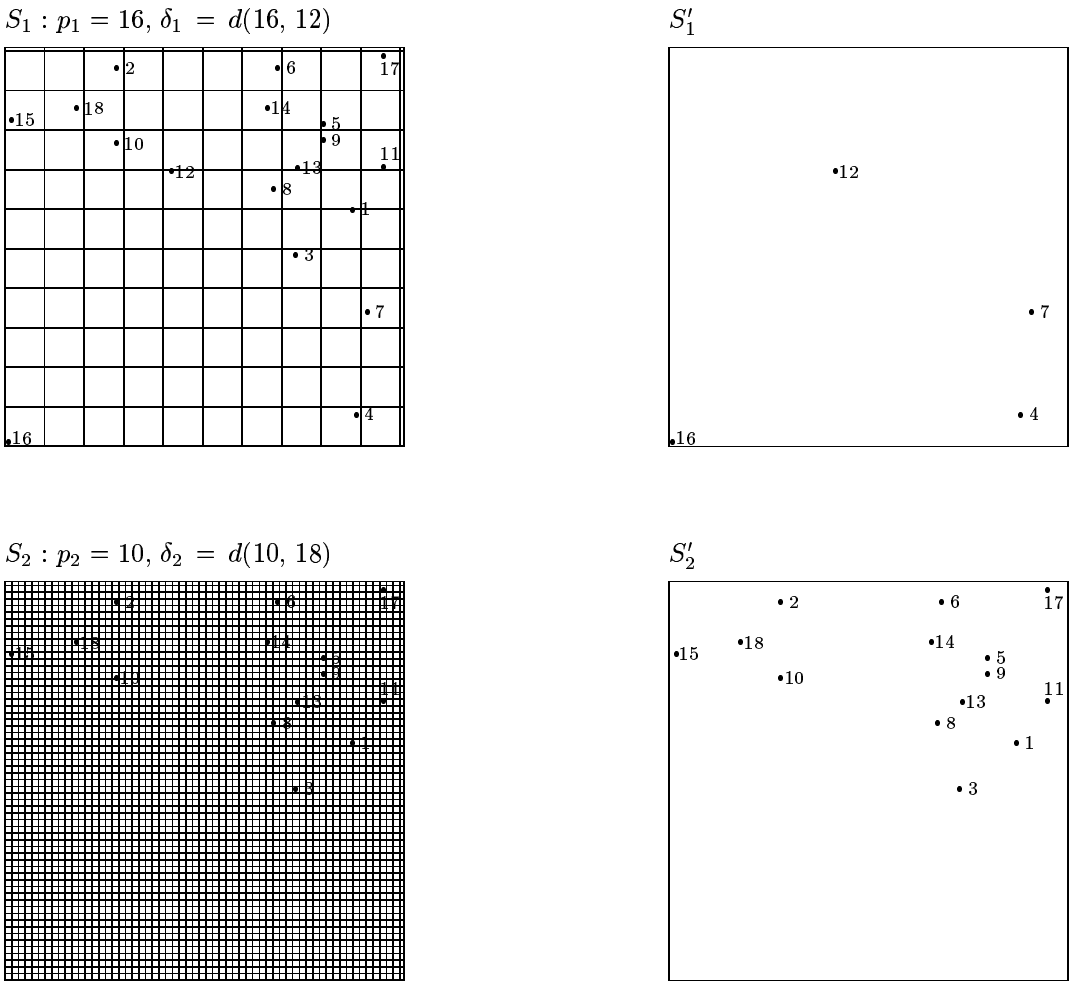

FIG. 4.3. *In this diagram a different point 18 is added to the first sparse partition. This point 18 is not sparse; it has neighbors 2, 15 and 10. In fact, because of the insertion of 18, these three points, which used to be sparse, are no longer sparse and are therefore placed in down*$_1$*. When the algorithm reached* S_2 *it then decided, probabilistically, that 10 should be the new pivot of that level. After making 10 the pivot it found that all points were sparse and so terminated the algorithm.*

---

fore 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 4.2. *Assume algorithm Insert*(q) *has computed 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 occurs, 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 3.2 and 2.3, 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.

It remains to show that $N_i(p, \widetilde{S}_i) = \emptyset \iff p \in \widetilde{S}'_i$ for any $p \in \widetilde{S}_i$, see Definition 3.2, 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 3.6 and therefore, for any $p \in D_i$,

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

If $p = q$, then $N_i(p, \widetilde{S}_i) = \emptyset \iff q \in \widetilde{S}'_i$ 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 $\widetilde{S}_{i+1} \cup \widetilde{S}'_i = (S_{i+1} \cup S'_i) \cup down_{i-1} \cup \{q\} = S_i \cup down_{i-1} \cup \{q\}$ 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 4.3. *At the end of step* **3** *of algorithm Insert, we have computed a uniform sparse partition for $\widetilde{S}$ according to Definitions 3.2 and 2.3.*

Proof. Refer to Figure 4.4. Let $\hat{\imath}$ denote the value of i after the last completion of step **3**. This in particular means that for each level $1 \leq j < \hat{\imath}$, no rebuilding has yet occurred 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 4.2.)

Invariant $\mathsf{INS}(\hat{\imath})$ (a) implies that the 5-tuples at levels $1, \ldots, \hat{\imath} - 1$ satisfy Definitions 3.2 and 2.3. Now, the last iteration at level $\hat{\imath}$ is either a rebuilding or produces a 5-tuple such that $q \in \widetilde{S}'_i$.

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

In the latter case, another application of Lemma 4.2 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 3.2 and 2.3. 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 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, only the new point q can make a previously sparse point non-sparse. We express this in the following lemma.

LEMMA 4.4. *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 (4.1). The second claim is true because $p \in down_j$ implies $p \in S \setminus S_{j+1}$, and by Lemma 3.6, $N_j(p, S) = \emptyset$ for each $p \in S \setminus S_{j+1}$. □

LEMMA 4.5. *Let the sets $down_0, \ldots, down_i$ be as defined in Equation (4.1). 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 4.4. 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 3.2 and 3.3. 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 4.4, 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 2.2 (2), Lemma 3.4 (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 $\Psi \in \{-1, 0, 1\}^D$, there is at most one point p in S which satisfies $p \in down_j \wedge p \in b_j^\Psi(q)$. □

Computing the *down* sets in constant time. We have just shown that the total size of the *down* sets is constant, implying in particular that 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 (4.1), 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 $(S_i, S_i', p_i, q_i, \delta_i)$ to $(\widetilde{S}_i, \widetilde{S}_i', \widetilde{p}_i, \widetilde{q}_i, \widetilde{\delta}_i)$ 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 take care of 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 and 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. $\mathcal{L}(b)$) 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. (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 a point is only deleted from a non-sparse set S_{i+1} during the insertion algorithm if a rebuilding occurs.

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 4.6. *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 occurs, 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 $\mathrm{E}(T_i) = O(|S_i|)$ in this case. Otherwise, $\mathrm{E}(T_i) = O(1)$. 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 so obtain that the expected value of T_i is $O(1)$, independently of the previous state of the data structure.

We now note that $\mathrm{E}(\sum_{i=k}^{n} T_i) = O(|S_k|)$ for any $1 \le k \le n$. This is because either a rebuild occurs at level k, requiring $O(|S_k|)$ expected time, and we are done. Otherwise $\mathrm{E}(T_k) = O(1)$ and by induction, $\mathrm{E}(\sum_{i=k+1}^{n} T_i) = O(|S_{k+1}|)$. Since $|S_{k+1}| \le |S_k| - 1$ (the pivot at level N is always sparse) we can choose the constant within the big-oh large enough to conclude that $\mathrm{E}(\sum_{i=k}^{n} T_i) = O(|S_k|)$ in this case as well.

Let $N = \lceil \log n \rceil$. From the above discussion:

$$\begin{aligned} \mathrm{E}(X) &\le \sum_{i=1}^{N} \mathrm{E}(T_i) + \mathrm{E}\Big(\sum_{i=N+1}^{n} T_i\Big) \\ &\le O(\log n) + O(\mathrm{E}(|S_{N+1}|)) = O(\log n) \end{aligned}$$

since $\mathrm{E}(|S_{\lceil \log n \rceil + 1}|)$ 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.

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 2.6 (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: Either there is a value $d_i^*(p)$ in H_i, or 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 4.5 and 4.6.

LEMMA 4.7. *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. Recall 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.

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 those restricted distances which can change due to the change of the sparse partition and which have not been handled by rebuilding have been treated by the procedures shown in Figures 4.5 and 4.6.

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 Lemma 4.1, 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.
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 which 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 3.7. Moreover, from the proof of Lemma 3.7 we know that the restricted distances can be computed by searching the area of at most $4D$ boxes away from p in the grids that store the sparse sets $S'_{i+l}, 0 \leq l \leq D$. 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. □

LEMMA 4.8. *Algorithm Insert(q) correctly maintains the data structure and takes expected time $O(\log n)$.*

Proof. From Lemma 4.2, steps **1-3** establish that $S \cup \{q\}$ is stored uniformly as a sparse partition. Also, from Lemma 4.7, the heaps are maintained correctly by step **4**. This proves the correctness of the algorithm.

As shown in Lemma 4.6, steps **1-3** have expected cost $O(\log n)$. Now consider step **4**. From Lemma 4.5 we know that the heap update procedures are only called for a constant number of points. Since, by Lemma 4.7, 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, and may involve some points becoming sparse at their current levels due to the deletion of q, thus causing them to move up a few levels. In particular, points which move to lower levels during an insertion of q move back to their old levels when q is deleted directly afterwards (provided no rebuilding takes place).

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, $1 \leq i \leq \ell$. 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.

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, as in the insertion algorithm. 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. 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.

The deletion algorithm starts at the top level and moves downward, as algorithm Insert. We define $up_0 := \emptyset$. Let $i \geq 1$. The following invariant, which is analogous to invariant $\mathsf{INS}(i)$ in algorithm Insert, holds if the algorithm attempts to construct the 5-tuple for $\widetilde{S}_i$ without having performed a rebuilding yet:

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 3.2 and 2.3.

(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 performed, 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 cause heap updates. The deletion algorithm is described in Figure 4.7.

From the similarity of invariants $\mathsf{INS}(i)$(b) and $\mathsf{DEL}(i)$(b), it is easy to see that the arguments used in Lemma 4.5 to derive the bound on the size of the *down* sets carry over to the *up* sets, and thus we obtain $|\bigcup_{1 \leq i \leq L} up_i| \leq 3^D$.

The computation of the *up* sets is slightly different from the computation of the *down* sets. In order to compute the *down* sets efficiently, we gave an alternative but equivalent definition for these sets (Equation (4.1)), and showed that the alternative definition could be efficiently realized. The difficulty there was that the points in $down_i$ could come from the set $S \setminus S_{i+1} = S'_1 \cup \ldots \cup S'_i$, and there seemed to be no direct way of extracting the points of $down_i$ from this set. 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 $\mathsf{DEL}(i)$(b), 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.

LEMMA 4.9. *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 4.10. *There exists a data structure which 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 use 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, that the floor function can be computed in constant time, and that the updates do not depend upon the random choices made by the data structure.*

5. An algebraic computation tree implementation. The solution from the previous section uses a somewhat inelegant model, which is an uneasy marriage of the unit-cost RAM and the algebraic computation tree. This algorithm may also be a poor one to use in practice, as the integers (box indices) which are computed as intermediate results may be so large that they cannot be manipulated in constant time by the hashing routines. Even if we implement the box dictionary by search trees, dividing point coordinates by very small numbers (interpoint distances) may lead to numerical problems.

We now present an algorithm which fits into the algebraic computation tree model. It can be verified that the algorithm requires only addition, subtraction, comparison and multiplication of real numbers to maintain the closest pair (although computing the actual value of the minimum distance $\delta(S)$ in the L_t metric for $1 < t < \infty$ will require the t-th root function as well). However, it is well known that the floor function used by the previous algorithm is very powerful: The maximum-gap problem requires $\Omega(n \log n)$ time in the algebraic computation tree model, but can be solved in $O(n)$ time by adding this function. Hence, we may expect some increase in the time complexity of the operations, and this does indeed turn out to be the case.

Note that the floor function was only used to compute the grid box containing a given point. Therefore, we will modify the algorithm Theorem 4.10 by using a degraded grid for which we only need algebraic functions. The method we use already appears in [9] and [11]. We sketch the structure here and refer to these papers or [18] 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 *explicitly*, 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 5.1.

The degraded grid can be maintained under insertions and deletions of points in logarithmic time, and the box containing a point can be identified in logarithmic time as well.

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 3.2. 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 which 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 $\widehat{V} \subseteq V$. This is because 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.

LEMMA 5.1. *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') $q \in N_{\delta,V}(p) \iff p \in N_{\delta,V}(q)$.

LEMMA 5.2. *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 (3.2) and (3.3) 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 (3.4) and (3.5), because these equations hold by the definition of the hyperplanes employed in the proof. See Figure 3.3. □

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 (3.1) for standard grids, we define

$$S'_i := \{p \in S_i : p \text{ sparse in } \mathcal{DG}_{g_i,S_i} \text{ relative to } S_i\}. \tag{5.1}$$

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 (5.1) yields a definition of a sparse partition analogous to the one given in Definition 3.2 for the grid case.

We adapt the abstract definition of the sparse partition (Definition 2.1) to degraded grids by changing "$\delta_i/2$" to "$\delta_i/4$" and "$\delta_i/4D$" to $\delta_i/16D$". The bounds in Lemma 2.2 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 3.3, using (N.1')-(N.3') instead of (N.1)-(N.3), we get

LEMMA 5.3. *Using the definition for S'_i given in Equation (5.1), we get a sparse partition according to Definition 2.1, 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 yet performed a rebuilding, 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

$$V_1 \supseteq V_2 \supseteq \ldots \supseteq V_L. \tag{5.2}$$

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 which 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. In particular, the update algorithms maintain the degraded grid $\mathcal{DG}_{\delta,S_i}$ for *both* S_i *and* S'_i. That is, a point p which 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 5.2. 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 (5.2) 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 3.5 and Lemma 3.6, obtained using (N.4'), are

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

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 (4.1). The corresponding statement to Lemma 4.4, which is obtained by using the above two statements, is

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

With these preparations, we can prove that Lemma 4.5 remains valid, i.e.

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

We recall the proof of Lemma 4.5 together with the changes that are needed. The proof is now done with (5.3) and (N.5') replacing Lemma 4.4 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 (5.3). Let $\Psi \in \{-1, 0, 1\}^D$ be a signature such that $p \in b_j^\Psi(q)$. Refer to Figures 3.2 and 3.3. 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 (5.3), $N_j^\Psi(q)$ 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 (5.3). 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 (5.2). 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. In Theorem 4.10, the box dictionary, which stores the indices of the non-empty grid boxes, was implemented using perfect hashing. Clearly, we can also store these indices in a balanced binary search tree. Since identifying the box containing a given point now takes $O(\log n)$ time anyway on a structure of size n, the cost of searching for that box in the box dictionary (also $O(\log n)$) may be ignored. As we are now doing in logarithmic time what previously took constant time, the overall running time is also increased by a logarithmic factor. We conclude:

THEOREM 5.4. *There exists a data structure which 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 $O(\log^2 n)$ expected time per update. The data structure is randomized and fits in the algebraic decision tree model. The time bounds are obtained assuming that the updates do not depend upon the random choices made by the data structure.*

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 no longer has the property that its distribution is independent of the history of updates. This does not affect the analysis of the algorithms, however.

6. Extensions. The data structure, as described so far, uses $O(n)$ *expected* space. 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 in that they will bound the expected running time of a sequence of update operations. We also show that this variant executes an update sequence quickly with high probability.

6.1. A data structure with linear space in the worst case. Recall that the space requirements are bounded by the sum of the sizes of the non-sparse sets $S_1, \ldots, S_L$ of the sparse partition, whose expected value was shown to be $O(n)$. To turn this into a worst case bound, our first step is to slightly modify the algorithm *Sparse_Partition* given in Section 2.

The modification is as follows: After picking the pivot of the grid 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 retain it as the pivot. Otherwise, we discard it and make a new random choice, continuing this process until a good pivot is found. We then continue on to the next set, making sure it has a good pivot as well, etc. Note that if all of the pivots in the sparse partition are good then, for all i, $|S_i'| \geq |S_i|/2$ so $|S_{i+1}| < |S_i|/2$, $\sum_{j>i} |S_j| = O(|S_i|)$ and the data structure will use $O(n)$ space. Furthermore, since $|S_{i+1}| \leq |S_i|/2$, the sparse partition has only $O(\log n)$ levels.

Note that at least half of the elements of a set are good pivots, and so at most two trials are needed on average until a good pivot is found. Using the same data structures as before to implement the sparse partition, we can easily prove that:

LEMMA 6.1. *Let S be a set of n points in $\mathbb{R}^D$. The modified version of algorithm Sparse_Partition produces a sparse partition for S of worst-case size $O(n)$ in $O(n)$ expected time.*

Note that the above lemma only discusses creating the sparse partition itself, and does not discuss creating the auxiliary data structures. This additional work can however be completed within the same time bound and we therefore do not discuss it. In the next section, in which we discuss high probability bounds, we will need to distinguish between constructing the sparse partition itself on the one hand and the complete data structure on the other. For the remainder of this section we assume that the procedures *Build* and *Near_Build* have been modified as above to produce sparse partitions of worst-case linear size.

We now move on to the update algorithms. The idea will be to maintain a sparse partition all of whose pivots are good, or at least not too far from being good. We make the following observations:

1. The uniformity property (Definition 2.3) 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 $2/n$ for a set of size n.
2. Updates can gradually unbalance the data structure in the sense that more than a constant proportion of the elements at that level can become non-sparse. In the earlier version of the algorithm the uniformity condition guaranteed that the data structure was, probabilistically, well balanced. Lacking the uniformity condition we enforce the balance of the sparse partition "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, rebuilding 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. This will ensure that $|S_i'|$ is always at least some constant proportion of $|S_i|$. In the sequel we call these rebuildings *amortized* to distinguish them from the *probabilistic* rebuildings that can be caused by a particular update.

We now sketch the modifications to the update algorithms, using the notation from Section 4. We associate two variables, $last_i$ and $count_i$, with level i of the current sparse partition. $last_i$ equals $|S_i|$ after the last (amortized or probabilistic) rebuilding that affected S_i, i.e. the last rebuilding performed at a level $j \leq i$. $count_i$ denotes the number of update operations since the last rebuilding that affected level i. In what follows, $c > 1$ is a sufficiently large constant.

In the insertion algorithm (Figure 4.4), let q be the newly-inserted point, and suppose we are at level i, $down_{i-1}$ has been computed, and $\widetilde{S}_i = S_i \cup down_{i-1} \cup \{q\}$. Only Step 2 (checking for rebuild) is changed as follows:

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

The first item is the amortized rebuilding discussed above, and the second item is one component of the probabilistic rebuilding in the original algorithm. Note that the first part of that rebuild step—which ensured the uniformity of the pivots by probabilistically deciding whether to make one of the elements in $down_{i-1} \cup \{q\}$ the new pivot—does not appear here. Since we do not have complete uniformity here anyway (bad elements cannot be pivots), we treat a newly inserted element as if it were a bad element.

In the deletion algorithm (Figure 4.7), suppose again that we are at level i, up_{i-1} has been computed and we have $\widetilde{S}_i = (S_i \setminus up_{i-1}) \setminus \{q\}$. Only Step 2 is changed as follows:

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

It is clear that the above modifications do not affect the correctness of the update algorithms. We now analyze its cost. Firstly note that immediately after a rebuilding $|S_i'| \geq |S_i|/2$. A single update may only add or subtract a constant number of items to or from S_i and S_i'. Thus, if c is taken to be large enough, the amortized rebuildings guarantee that $|S_i'| > |S_i|/4$ always holds, ensuring that (i) the data structure has $O(\log n)$ levels and (ii) the size of the structure is $O(n)$. (Note that the constant number of points that can be moved at each step depends upon the dimension D, so c must be chosen dependent upon D as well.)

As long as no rebuilding occurs, the algorithm uses a constant number of dictionary operations at each level. Since there are $O(\log n)$ levels the total expected cost of all dictionary operations is $O(\log n)$. The heap updates cost $O(\log n)$ time as before. Thus the total expected time required for an update operation which does not perform a rebuilding is $O(\log n)$.

Recall now that there are two types of rebuildings; amortized and probabilistic. We start by analyzing the amortized rebuildings. Note that an amortized rebuilding occurs on a set S_i of size m only if a sequence of $\Theta(m)$ updates has occurred without a

rebuilding of S_i. Since a rebuilding costs $O(m)$ expected time each of the sequence of $\Theta(m)$ updates incurs an $O(1)$ amortized cost for the rebuilding of S_i. Summing over all $O(\log n)$ levels in the data structure gives an amortized expected cost of $O(\log n)$ per update operation.

For the probabilistic rebuildings, we will show that for each update which affects a set S_i of size m, the probability of a rebuilding is at most c'/m for some fixed constant c'. Since the expected cost of a rebuilding is $O(m)$ this means that the expected cost of rebuilding a set at a given level at any particular step will be $O(1)$. Summing over all $O(\log n)$ levels yields a total expected cost of $O(\log n)$ per update.

To show that the probability of a rebuilding is at most c'/m for some c' we note that immediately after a rebuilding the probability of rebuilding a set S_i of size m is c'/m where $c' = 2$. Between rebuildings, an adversary gains knowledge about the possible pivots of the data structure permitting him to force a rebuilding with an appropriately chosen insert or delete command. However, we have already seen that an adversary can only exclude at most a constant number of points from being a pivot in each update operation. Thus, if c is taken to be large enough that after m/c updates at a level of initial size m, the adversary still must consider $|S_i|/4$ possible good pivots, this implies that the probability of a probabilistic rebuilding (i.e. item 2 of the modified updates) is still $O(1/m)$. This proves the following:

THEOREM 6.2. *The data structure of Theorem 4.10 can be modified to use $O(n)$ worst-case space. The modified data structure has $O(\log n)$ amortized expected update time, and the time complexities of all other operations are unchanged.*

Applying the same modifications to the data structure of Theorem 5.4 we get:

THEOREM 6.3. *The data structure of Theorem 5.4 can be modified to use $O(n)$ worst-case space. The modified data structure has $O(\log^2 n)$ amortized expected update time, and the time complexities of all other operations are unchanged.*

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 probability that the running time of the data structure of Theorem 6.3 on such an update sequence deviates significantly from its expectation. In what follows we say that an event occurs with *n-polynomial probability* if, for any fixed $s > 0$, it occurs with probability $1 - O(n^{-s})$. A process is said to take $O(f(n))$ time with n-polynomial probability if, for any fixed $s > 0$, the process takes at most $c(s)f(n)$ time for sufficiently large n, where $c(s)$ is a constant dependent upon s, with probability $1 - O(n^{-s})$.

THEOREM 6.4. *Let S be a set of n points in $\mathbb{R}^D$. The data structure of Theorem 6.3 performs a sequence of n updates on S in $O(n \log^2 n)$ time with n-polynomial probability.*

Remark: We can also prove that the data structure of Theorem 6.2 processes a sequence of $\Theta(n)$ updates, starting with a set of size n, in $O(n \log^2 n)$ time with n-polynomial probability; see [18] for details.

Proof. First note that the theorem is obviously correct if we only count the costs of the part of the updates that do not include rebuildings. The non-rebuilding part of the update consists of $O(1)$ heap operations, each costing $O(\log n)$ time for a total of $O(\log n)$ time, and of $O(\log n)$ dictionary operations each costing $O(\log n)$ time for a total of $O(\log^2 n)$ time. (Recall that our data structure has $O(\log n)$ levels in the worst case for a set of size n.) Hence, if no rebuilding occurs, an update takes $O(\log^2 n)$ time in the worst case.

We now analyze the rebuilding cost, and show that rebuilding a sparse partition for a set of size m will require $O(m \log n)$ time with n-polynomial probability. First note that, during the rebuilding of a level, the chance of a random pivot being a good one is at least $1/2$; the probability of not finding a good pivot after k trials is therefore at most 2^{-k} so, with n-polynomial probability, only $O(\log n)$ pivots need to be checked before finding a good one. At first sight, it appears as if even checking if a single pivot is good should take $O(m \log n)$ time, since checking to see if a point is sparse appears to require $O(1)$ queries to the box dictionary, each costing $O(\log n)$ time.

However, if we only want to check if a pivot is good, we can avoid queries to the box dictionary as follows: During the building of the degraded grid data structure, we 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. (This faster method was not mentioned earlier as computing the sparse points of a set was never a bottleneck previously.) Thus we can identify the sparse points and decide whether the pivot is good in $O(m)$ time; it is only in the computing of the restricted distances where we will need $\Theta(m \log n)$ time.

We conclude that a good pivot can be found, and the grid for that level built and processed in $O(m \log n)$ time with n-polynomial probability. Since the levels decrease geometrically in size, good pivots for all succeeding levels of the partition can also be found in $O(m \log n)$ time with n-polynomial probability. The remainder of the *Build* procedure takes $O(m \log n)$ time as before.

We now analyze the amortized and probabilistic rebuildings separately, studying the amortized ones first. Actually, using the same reasoning as developed in the previous subsection, we find that an amortized rebuilding of set S_i of size m is only performed if $\Theta(m)$ previous updates did not rebuild S_i. Now, rebuilding S_i requires $O(m \log n)$ time with n-polynomial probability, so, with n-polynomial probability, the amortized rebuilding cost per update of S_i is $O(\log n)$. Since there are $O(\log n)$ levels, this adds up to $O(n \log^2 n)$ time with n-polynomial probability.

Before analyzing the cost of the probabilistic rebuildings, i.e., the rebuildings caused by a deletion of a pivot or insertion/deletion of the nearest neighbor to a pivot, it will help to quickly review what we are trying to study. As we start with a data structure which contains n points and then make n updates to it, the data structure contains at most $2n$ items at any update step. Since $|S_i'| > |S_i|/4$ we have $|S_{i+1}| < 3|S_i|/4$ so the data structure never contains more than $L = \log_{4/3}(2n) = \Theta(\log n)$ levels. Now suppose that $|S_i| = m$. Then, as described in the previous subsection, S_i is rebuilt with probability $O(1/m)$. The cost of rebuilding S_i will be $O(m \log n)$ with n-polynomial probability. Let $m_{i,t}$ denote the size of S_i at the end of update $t-1$, for $1 \leq i \leq L$ and $1 \leq t \leq n$. If S_i does not exist at the end of update $t-1$ set $m_{i,t} = 1$ by convention. Now define the random variables

$$X_{i,t} = \begin{cases} m_{i,t} & \text{with probability } 1/m_{i,t} \\ 0 & \text{with probability } 1 - 1/m_{i,t} \end{cases}$$

Then the cost of probabilistically rebuilding S_i at update t, counting both degraded grid and heap operations, is $O(X_{i,t} \log n)$ with n-polynomial probability by the above

discussion. Hence the total cost of probabilistic rebuildings over all updates is bounded by $O(M \log n)$ with n-polynomial probability, where $M = \sum_{i,t} X_{i,t}$. We now show that M itself is $O(n \log n)$ with n-polynomial probability. The proof of Theorem 6.4 will follow.

The first complication we encounter in bounding M is that the $X_{i,t}$ are *not* independent because the $m_{i,t}$ depend upon each other. Nevertheless, since $|S| \leq 2n$ and the amortized rebuilding ensures that $|S_{i+1}| < 3|S_i|/4$, the relation

$$m_{i,t} \leq \max\{1, 2 \cdot (3/4)^{i-1} \cdot n\} \tag{6.1}$$

must always hold. We sidestep the dependence issue by showing that, for *any* values of $m_{i,t}$ that satisfy Equation (6.1), M has n-polynomial probability of being $O(n \log n)$. This allows us to assume that we are given some *fixed* (but arbitrary) values $m_{i,t}$ which satisfy (6.1), and that the variables $X_{i,t}$ are as defined above and *independent* of each other.

The following lemma is obtained by a straightforward modification of the proof of the Chernoff bound given in [14, p.68]:

LEMMA 6.5. *Let $Y_1, \ldots, Y_k$ be independent random variables, and let $a_1, \ldots, a_k \in [1, A]$ for some $A \geq 1$. For $i = 1, \ldots, k$ suppose that*

$$Y_i = \begin{cases} a_i & \text{with probability } 1/a_i \\ 0 & \text{with probability } 1 - 1/a_i \end{cases}$$

and let $Y = \sum_{i=1}^{k} Y_i$. Then, for any $\delta > 0$

$$\Pr[Y > (1+\delta)k] < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{k/A}. \tag{6.2}$$

Proof. Let $\lambda = (\ln(1+\delta))/A$ and note that $\lambda > 0$. Then:

$$\begin{aligned} \Pr[Y > (1+\delta)k] &= \Pr[e^{\lambda Y} > e^{\lambda(1+\delta)k}] \\ &\leq \frac{\mathrm{E}(e^{\lambda Y})}{e^{\lambda(1+\delta)k}} \\ &= \frac{\prod_{i=1}^{k} \mathrm{E}(e^{\lambda Y_i})}{e^{\lambda(1+\delta)k}} \\ &= \frac{\prod_{i=1}^{k} \left(\frac{1}{a_i} e^{\lambda a_i} + 1 - \frac{1}{a_i}\right)}{e^{\lambda(1+\delta)k}}. \end{aligned}$$

However, $f(x) = e^{\lambda x}/x + 1 - 1/x$ can easily be seen to be an increasing function of x, for $x > 0$. Therefore:

$$\Pr[Y > (1+\delta)k] \leq \frac{\left(\frac{1}{A} e^{\lambda A} + 1 - \frac{1}{A}\right)^k}{e^{\lambda(1+\delta)k}} \leq \frac{(1+\delta/A)^k}{(1+\delta)^{(1+\delta)k/A}} < \left[\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right]^{k/A}. \quad \square$$

Let s be a given positive integer. We apply Lemma 6.5 to $X_i = \sum_{t=1}^{n} X_{i,t}$ for each $1 \leq i \leq 5 \log\log n$, noting that $m_{i,t} \leq 2n$ for this range of i. By choosing $\delta = c \cdot \log n / \log\log n - 1$ for sufficiently large $c > 0$ and $A = 2n$, we obtain that, for all $1 \leq i \leq 5 \log\log n$:

$$\Pr[X_i > c \cdot n \cdot \log n / \log\log n] < n^{-s-1} \tag{6.3}$$

We now apply Lemma 6.5 to $X_i = \sum_{t=1}^{n} X_{i,t}$ for each $5 \log\log n < i \leq L$, noting that $m_{i,t} \leq n/\log n$ for this range of i. By choosing $\delta = c' - 1$ for sufficiently large $c' > 1$ and $A = n/\log n$, we obtain that, for all $5 \log\log n < i \leq L$:

$$\Pr[X_i > c' \cdot n] < n^{-s-1} \tag{6.4}$$

From (6.3) and (6.4) we conclude that $M = \sum_{i=1}^{n} X_i = O(n \log n)$ with probability at least $1 - n^{-s}$, and hence $M = O(n \log n)$ with n-polynomial probability. □

7. Concluding remarks. In this paper, we have given the first solution to the fully-dynamic closest pair problem which achieves linear space and polylogarithmic update time simultaneously. After a preliminary version of this paper was published, Kapoor and Smid [13] achieved this goal with a deterministic method which 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 case $D = 2$, where ℓ is an arbitrary non-negative integer constant. Also, Callahan and Kosaraju [4] gave a deterministic data structure achieving $O(\log^2 n)$ update time while using linear space, for any fixed dimension. Finally, Bespamyatnikh [3] gave an optimal deterministic solution for the dynamic closest pair problem: for any fixed dimension, his technique achieves $O(\log n)$ update time and uses $O(n)$ space.

REFERENCES

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

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

[3] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages 152–161, 1995.

[4] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest-pair and n-body potential fields. In *Proc. 6th ACM-SIAM Sympos. Discrete Algorithms*, pages 263–272, 1995.

[5] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comput. System Sci.*, 18:143–154, 1979.

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

[7] M. T. Dickerson, R. L. Drysdale, and J. R. Sack. Simple algorithms for enumerating interpoint distances and find ing k nearest neighbors *Internat. J. Comput. Geom. Appl.*, 2:221–239, 1992.

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

[9] A. Datta, H.-P. Lenhof, C. Schwarz, and M. Smid. Static and dynamic algorithms for k-point clustering problems. *J. Algorithms*, 19:474–503, 1995.

[10] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert and R.E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.*, 23:738–761, 1994.

[11] M. J. Golin, R. Raman, C. Schwarz, and M. Smid. Simple randomized algorithms for closest pair problems. *Nordic Journal of Computing*, 2:3–27, 1995.

[12] S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118: 34–37, 1995.

[13] S. Kapoor and M. Smid. New techniques for exact and approximate dynamic closest-point problems. *SIAM J. Comput.*, 25:775–796, 1996.

[14] R. Motwani and P. Ragahavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[15] W. Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees, Communications of the ACM, 33(6), pages 668-676, 1990.

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

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

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

[19] R. Seidel and C. R. Aragon , Randomized search trees. *Algorithmica*, 16:464–497, 1996.

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

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

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

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

[24] 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.

(1) **Algorithm** Insert (q);
(2) **begin**
(3) **1. initialize:** $i := 1$; $down_0 := \emptyset$; $h := \infty$
(*) From invariant 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:
(4) flip an $|\widetilde{S}_i|$-sided coin, giving pivot $\widetilde{p}_i$ of $\widetilde{S}_i$;
(5) **if** $\widetilde{p}_i \notin S_i$ **then**
(6) $Near\_Build(\widetilde{S}_i, \widetilde{p}_i, i)$; $h := i - 1$; goto **4.**
(7) **else**
(8) the old pivot p_i of S_i is also the pivot for $\widetilde{S}_i$
(9) **fi**;
(10) **if** $d(p_i, p) < \delta_i$ for some $p \in down_{i-1} \cup \{q\}$ **then**
(11) $Build(\widetilde{S}_i, i)$; $h := i - 1$; goto **4.**
(12) **else**
(13) do nothing (* $d_i = d(p_i, S_i) = d(p_i, \widetilde{S}_i)$ *)
(14) **fi**;
3. Determine $\widetilde{S}'_i$:
(15) compute the set $down_i$ from its candidate set $D_i = S'_i \cup down_{i-1}$, see Eq. (4.1);
(16) $\widetilde{S}'_i := D_i \setminus down_i$; $\widetilde{S}_{i+1} := S_{i+1} \cup down_i$; (* now $\widetilde{S}_{i+1} = (\widetilde{S}_i \setminus \widetilde{S}'_i) \setminus \{q\}$ *)
(17) **if** $N_i(q, \widetilde{S}_i) = \emptyset$ **then**
(18) add q to $\widetilde{S}'_i$; goto **4.** (* q is sparse in $\widetilde{S}_i$, and so $\widetilde{S}_{i+1} = S_{i+1}$ *)
(19) **fi**; (* q is not sparse in $\widetilde{S}_i$ *)
(20) add q to $\widetilde{S}_{i+1}$;
(21) $i := i + 1$; 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 performed.
Otherwise, $h = \infty$ and so $\min\{i, h\} = i$. *)
(22) **for** $\ell := 1$ **to** $\min\{i, h\}$ **do**
(23) **forall** $p \in down_\ell \setminus down_{\ell-1}$ **do**
(24) removefromheap(p, ℓ)
(25) **od**
(26) **forall** $p \in down_{\ell-1} \setminus down_\ell$ **do**
(27) addtoheap(p, ℓ)
(28) **od**;
(29) **od**;
(30) **if** $q \in \widetilde{S}'_{\min\{i,h\}}$ **then**
(31) addtoheap$(q, \min\{i, h\})$
(32) **fi**;
(33) **end**;

FIG. 4.4. *Algorithm Insert(q). The heap update procedures addtoheap and removefromheap called in step* **4** *will be given later.*

```
(1)   proc removefromheap (p, h);
(2)    begin
(3)        (* p starts moving at level i, i.e. p ∈ S'_i, but p ∉ S̃'_i *)
(4)        DELETE(H_i, it(p));
(5)        for ℓ := i to min{i + D, h} do
(6)            forall r ∈ S'_ℓ ∩ S̃'_ℓ such that d(r, p) < δ_ℓ do
(7)                CHANGE_KEY(it(r), d*_ℓ(r))
(8)            od
(9)        od;
(10)   end;
```

FIG. 4.5. *Procedure removefromheap*(p, h).

```
(1)   proc addtoheap (p, h);
(2)    begin
(3)        (* p stops moving at level j, i.e. p ∉ S'_j, but p ∈ S̃'_j *)
(4)        compute d*_j(p); let r be such that d*_j(p) = d(r, p) if it exists;
(5)        it(p) := new item; it(p).key := d*_j(p); it(p).point := p; it(p).point2 := r;
(6)        INSERT(H_j, it(p));
(7)        for ℓ := j to min{j + D, h} do
(8)            forall r ∈ S'_ℓ ∩ S̃'_ℓ such that d(r, p) < δ_ℓ do
(9)                CHANGE_KEY(it(r), d*_ℓ(r))
(10)           od
(11)       od;
(12)   end;
```

FIG. 4.6. *Procedure addtoheap*(p, h).

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

FIG. 4.7. *Algorithm Delete*(q).

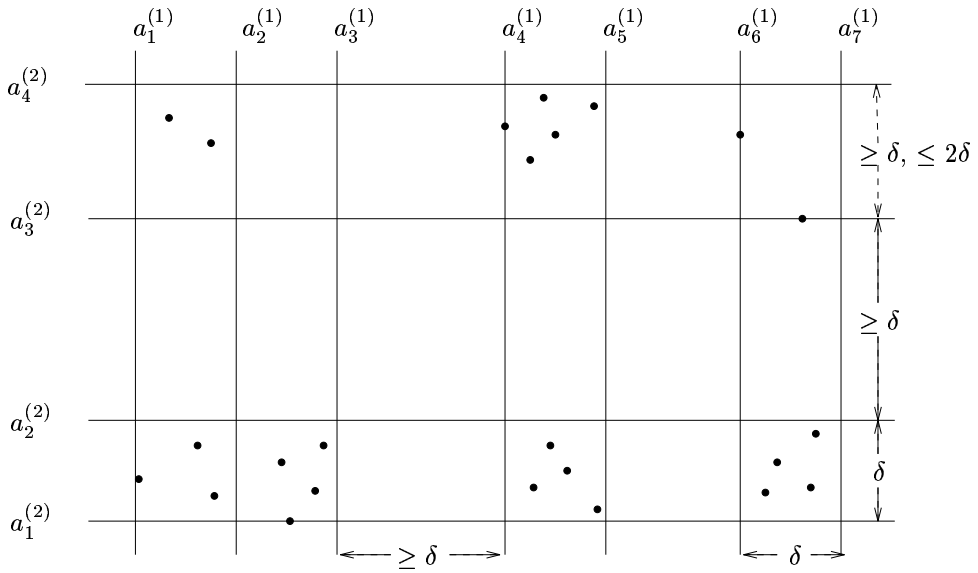

FIG. 5.1. *Example of a degraded δ-grid. It is dependent on the set stored in it.*