An optimal algorithm for closest pair maintenance

Sergei N. Bespamyatnikh
Department of Mathematics and Mechanics,
Ural State University,
51 Lenin St., Ekaterinburg 620083, Russia.
e-mail: Sergei.Bespamyatnikh@usu.ru.

Abstract

Given a set S of n points in k-dimensional space, and an L_t metric, the dynamic closest pair problem is defined as follows: find a closest pair of S after each update of S (the insertion or the deletion of a point). For fixed dimension k and fixed metric L_t, we give a data structure of size $O(n)$ that maintains a closest pair of S in $O(\log n)$ time per insertion and deletion. The running time of algorithm is optimal up to constant factor because $\Omega(\log n)$ is a lower bound, in algebraic decision-tree model of computation, on the time complexity of any algorithm that maintains the closest pair (for $k = 1$). The algorithm is based on the fair-split tree. The constant factor in the update time is exponential in the dimension. We modify the fair-split tree to reduce it.

1 Introduction

The dynamic closest pair problem is one of the very well-studied proximity problem in computational geometry [6, 17–20, 22, 24–26, 28–31]. We are given a set S of n points in k-dimensional space, $k \geq 1$, and a distance metric L_t, for $1 \leq t \leq \infty$. The point set is modified by insertions and deletions of points. Each point p is given as a k-tuple of real numbers $(p_1, \ldots, p_k)$.

The closest pair of S is a pair (p, q) of distinct points $p, q \in S$ such that the distance between p and q is minimal. The dynamic closest pair problem is defined as follows: find a closest pair (any) of S after each update of S.

We assume that the dimension k and the distance metric L_t are fixed. We use $d(p, q)$ to denote the distance between p and q.

A survey can be found in Schwarz's Ph.D.Thesis [24]. For the static closest pair problem and dimension $k = 2$, Shamos and Hoey [23] gave an algorithm with running time of $O(n \log n)$. Shortly after that, Bentley and Shamos [5] obtained this result for general dimension $k \geq 2$. In the *on-line* closest pair problem only insertions are allowed. For this problem Smid [28] obtained a data structure of size $O(n)$ that supports insertions in $O(\log^{k-1} n)$ amortized time. Schwarz, Smid and Snoeyink [26] presented a data structure of size $O(n)$ that maintains the closest pair in $O(\log n)$ amortized time per insertion.

Several algorithms are obtained for the dynamic closest pair problem [19, 20, 22, 24, 29–31]. In [20, 22, 29] the problem is solved with $O(\sqrt{n} \log n)$ update time using $O(n)$ space. In [19] Kapoor and Smid gave data structures of size $S(n)$ that maintain the closest pair in $U(n)$ amortized time per update, where for $k \geq 3$, size $S(n) = O(n)$ and time $U(n) = O(\log^{k-1} n \log\log n)$; for $k = 2$, size $S(n) = O(n \log n/(\log\log n)^m)$ and time $U(n) = O(\log n \log\log n)$; for $k = 2$, size $S(n) = O(n)$ and time $U(n) = O(\log^2 n/(\log\log n)^m)$ (m is an arbitrary non-negative integer constant). In [6] the author obtained an algorithm with $O(\log^{k+1} n \log\log n)$ update time and $O(n \log^{k-2} n)$ space. Callahan and Kosaraju [13] developed a tree-maintenance technique to solve a general class of dynamic problems. This technique can be used to maintain the closest pair in $O(\log^2 n)$ time and $O(n)$ space.

We give a linear size data structure that maintains the closest pair in $O(\log n)$ time per update. The algorithm is deterministic and the update time is worst-case. The algorithm fits in the algebraic computation tree model. In the algebraic computation tree model, there is a lower bound of $\Omega(n \log n)$ on the time complexity of any algorithm that solves the static closest pair problem for dimension $k = 1$ [3, 21]. So the running time of our algorithm is optimal up to a constant factor.

Our algorithm is based on the following idea. We use a hierarchical subdivision of space into boxes. Several proximity algorithms build hierarchical subdivisions of space [33, 15, 14, 28, 25, 24, 2, 12, 13]. These subdivisions differ by the shape of boxes, the overlap allowance, the manner of box splitting, the number of points in a box stored at a leaf. Our algorithm maintains almost cubical boxes. The boxes are split by almost middle cutting [7] which is similar to fair split [12, 13, 11]. Any smallest box contains exactly one of the given points. For each point we store some neighbor points. The closest pair is one of these pairs. To maintain efficiently these pairs we apply the dynamic trees of Sleator and Tarjan [27]. To insert a point we implement the point location. The point location also uses the dynamic trees. The idea to use dynamic trees for point location in hierarchical subdivisions is due to Cohen and Tamassia [15] and Chiang, Preparata and Tamassia [14]. Schwarz [24] applied the dynamic trees for the on-line closest pair problem and obtained an algorithm with worst-case $O(\log n)$ time per insertion and $O(n)$ space. Our hierarchical subdivision is similar to the *box decomposition* of [1] and the *fair-split tree* of [13]. In [13, 1] the point location uses the *topology tree* of Frederickson [16]. The topology tree is based on dynamic trees of Sleator and Tarjan [27].

In Section 2 we describe the fair-split tree. In Section 3 we show how to maintain the fair-split tree (without point location). Section 4 explains how to maintain neighbor information of points and the closest pair. In Section 5 we briefly recall the dynamic trees. In Section 6 we show how to implement the search on the dynamic trees. In Section 7 we discuss how to reduce the constant factors in the update time. Finally, in Section 8 we give some concluding remarks.

2 The fair-split tree

The fair-split tree is a hierarchical subdivision of space into boxes. We define a box to be the product $[a_1, a_1') \times \ldots \times [a_k, a_k')$ of k semiclosed intervals. The i-th side of this box is the interval $[a_i, a_i')$. If all sides have the same length, we say that the box is a *k-cube*. The cubes are useful in some proximity algorithms (for example, the all-nearest-neighbors algorithm of Vaidya [32, 33]). Unfortunately we cannot directly use cubes in a subdivision of space for the dynamic problem, because splitting a cube by a hyperplane $x_i = const$ does not give cubes. Another way is the using of the almost cubical boxes [7] and a fair-split [12, 13, 11, 10] or an almost middle cut [7]. The almost middle cut is similar to the fair split (but there is a difference in the definitions). In this paper, for the split of boxes, we use the definition of [7] but we shall call it the *fair split*. The fair-split tree is also applied to other dynamic problems [10, 8, 9].

The constant factors in the update and query time are exponential in the dimension. To decrease the constant factors we generalize the fair split by introducing a *separator* $s > 1$. In fact both the fair split [12, 13, 11] and the almost middle cut [7] use the separator that is equal 2. We establish geometric criteria for the fair split with separator to be suitable for maintenance of the fair-split tree. The separator must be at least Golden Ratio $\frac{\sqrt{5}+1}{2} \approx 1.61803$.

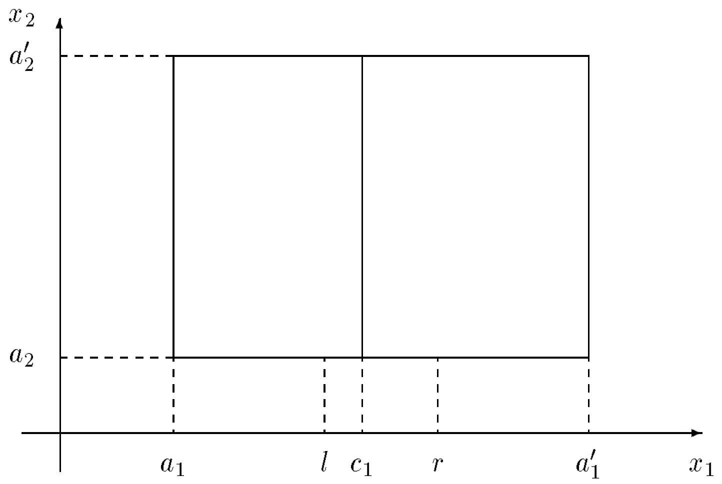

Figure 1: The hyperplane $x_1 = c_1$ determines a fair split of the box $[a_1, a_1') \times [a_2, a_2')$ if and only if $c_1 \in [l, r]$ where $l = \frac{sa_1 + a_1'}{s+1}$ and $r = \frac{a_1 + sa_1'}{1+s}$.

Definition 2.1 Let $[a, a')$ be an interval in $\mathbf{R}$ and b be a point in this interval. The split of the interval into the intervals $[a, b)$ and $[b, a')$ is a *fair split* if the length of larger interval is at most s times the length of smaller interval, i.e.

$$\frac{b-a}{a'-b} \in [\frac{1}{s}, s].$$

Definition 2.2 Let $B = [a_1, a_1') \times \ldots \times [a_k, a_k')$ be a box and $c_i \in (a_i, a_i')$ be a real number for some i. The split of B by the hyperplane $x_i = c_i$ into the boxes $B \cap \{x | x_i < c_i\}$ and $B \cap \{x | x_i \geq c_i\}$ is a *fair split* of B if the split of the interval $[a_i, a_i')$ by c_i is fair split.

The fair-split operation generates a relation on the set of boxes.

Definition 2.3 Let A and B be k-dimensional boxes. The box A is said to be an *s-sub-box* of B if A can be constructed from B by applying a (possibly empty) sequence of fair splits. We shall write $B \rightsquigarrow A$. For $k = 1$, we shall say that A is an *s-sub-interval* of B.

In fact the relation of s-sub-box is the product of s-sub-interval relation.

Proposition 2.4 *Let $A = [a_1, a_1') \times \ldots \times [a_k, a_k')$ and $B = [b_1, b_1') \times \ldots \times [b_k, b_k')$ be k-dimensional boxes. The box A is s-sub-box of B if and only if, for $i = 1, \ldots, k$, the interval $[a_i, a_i')$ is s-sub-interval of $[b_i, b_i')$.*

We now give another definition of s-sub-interval, and show that it is equivalent to that of Definition 2.3.

Definition 2.5 Let $[a, a')$ and $[b, b')$ be intervals in $\mathbf{R}$. Let $[a, a')$ is the sub-interval of $[b, b')$, i.e. $b \leq a < a' \leq b'$. The interval $[a, a')$ is called an *s-sub-interval* of the interval $[b, b')$ if one of the following conditions holds

1. $[a, a') = [b, b')$, or
2. $a = b$ and $|a' - a| \leq \frac{s}{s+1}|b' - b|$, or
3. $a' = b'$ and $|a' - a| \leq \frac{s}{s+1}|b' - b|$, or
4. $|a' - b| \leq \frac{s}{s+1}|b' - b|$ and $|a' - a| \leq \frac{s}{s+1}|a' - b|$, or
5. $|b' - a| \leq \frac{s}{s+1}|b' - b|$ and $|a' - a| \leq \frac{s}{s+1}|b' - a|$.

This Definition allows us to retrieve a sequence of fair cuts for two boxes A and B if $B \rightsquigarrow A$. The following Theorem gives the condition for the separator s when Definitions 2.3 and 2.5 are equivalent.

Theorem 2.6 *Definitions 2.3 and 2.5 define the same relation of s-sub-interval if and only if the separator is at least Golden Ratio, i.e $s \geq \frac{\sqrt{5}+1}{2} > 1.61803$*

Proof. For convenience we define the intermediate notion of a *one-sided s-sub-interval.* The interval $[a, a')$ is called a *one-sided s-sub-interval* of the interval $[b, b')$ if either the second or third condition of Definition 2.5 holds. Note that the conditions 4 and 5 are the combinations of two one-sided s-sub-intervals (for different sides).

Suppose that Definitions 2.3 and 2.5 define the same relation of s-sub-interval. Consider two intervals $[a, a')$ and $[b, b')$ such that $[a, a')$ is s-sub-interval of $[b, b')$ and $a = b$. The interval $[a, a')$ can be constructed by applying a sequence of N fair splits of $[b, b')$. It is clear that

$$\frac{|a' - a|}{|b' - b|} \in [\frac{1}{(s+1)^N}, \left(\frac{s}{s+1}\right)^N].$$

The maximal value of $a' - a$ after one fair split is at least the minimal value of $a' - a$ after two fair splits (by condition 2 of Definition 2.5), i.e.

$$\left(\frac{s}{s+1}\right)^2 \geq \frac{1}{s+1}.$$

Using $s > 1$ we get $s \geq \frac{\sqrt{5}+1}{2}$.

Let $s \geq \frac{\sqrt{5}+1}{2}$. Similarly we can show that any one-sided s-sub-interval is s-sub-interval (in term of Definition 2.3). Hence any pair intervals satisfying Definition 2.5 satisfy Definition 2.3. To prove inverse statement we show that the combination of three one-sided s-sub-intervals can be represented as a combination of two one-sided s-sub-intervals.

Let $[b, c')$ be a one-sided s-sub-interval of $[b, b')$, $[c, c')$ be a one-sided s-sub-interval of $[b, c')$ and $[c, d')$ be a one-sided s-sub-interval of $[c, c')$.

$$|d' - b| \leq |c' - b| \leq \frac{s}{s+1}|b' - b|$$

$$\frac{|d' - c|}{|d' - b|} = \frac{|c' - c| - |c' - d'|}{|c' - b| - |c' - d'|} \leq \frac{\frac{s}{s+1}|c' - b| - \frac{s}{s+1}|c' - d'| - \frac{1}{s+1}|c' - d'|}{|c' - b| - |c' - d'|}$$
$$= \frac{s}{s+1} - \frac{|c' - d'|}{(s+1)|d' - b|} < \frac{s}{s+1}$$

Hence $[c, d')$ is an s-sub-interval of $[b, b')$. □

The constant factors in the update time depend on the separation as $((s+2)(s+1))^k$. Decreasing the separator reduces these factors.

We do not include the condition of almost cubical boxes into the definition of the fair split of boxes although we shall apply fair split only for such boxes. The almost cubical boxes can be obtained from cubes by repeatedly applying a fair split by a hyperplane perpendicular to one of the longest side of box.

Definition 2.7 Let B be a box with sides $s_1, \ldots, s_k$. The box B is said to be an *s-box* if, for any $i, j \in \{1, \ldots, k\}$,

$$\frac{s_i}{s_j} \in [\frac{1}{1+s}, 1+s].$$

The fair-split tree is a binary tree T. With each node v of the tree T, we store a box $B(v)$ and a shrunken box $SB(v)$. The boxes satisfy the following conditions.

1. For any node v, the boxes $B(v)$ and $SB(v)$ are s-boxes.

2. For any node v, the box $SB(v)$ is an s-sub-box of $B(v)$.

3. For any node v, $SB(v) \cap S = B(v) \cap S$.

4. If v has two children u and w, then boxes $B(u)$ and $B(w)$ are the results of a fair split of the box $SB(v)$.

5. If v is a leaf, then $|S \cap B(v)| = 1$ and $SB(v) = B(v)$.

For a point $p \in S$ corresponding to the leaf v, let $B(p)$ denotes the box $B(v)$.

Let $parent(v)$, $lson(v)$, and $rson(v)$ denote parent, left son, and right son of the node v of T.

We use $d_{\min}(X,Y)$ to denote the distance between two sets $X, Y \subset \mathbf{R}^k$, i.e. distance $d_{\min}(X,Y) = \inf\{dist(x,y) | x \in X, y \in Y\}$. $d_{\max}(X,Y)$ denotes the maximal distance between two sets $X, Y \subset \mathbf{R}^k$, i.e. distance $d_{\max}(X,Y) = \sup\{dist(x,y) | x \in X, y \in Y\}$. $d(X)$ denotes the diameter of a set X, i.e. distance $d(X) = d_{\max}(X,X)$.

3 The maintenance of the fair-split tree

In this Section we shall show how to maintain the fair-split tree T under insertions and deletions of points. The deletion is simpler than insertion and we consider the deletion first.

Let p be a point to be deleted. Let us w be a leaf corresponding p, i.e. point $p \in B(w)$, v be the parent of w and $u \neq w$ be the sibling of v. We consider 2 cases.

1) u is a leaf (see Fig. 2 a)). Then set $SB(v) = B(v)$ and delete the leaves u and w.

2) u is an internal node (see Fig. 2 b)). Then delete the node w, set $B(u) = B(v)$, and collapse the edge (u, v), i.e. set $parent(u) = parent(v)$, delete the node v, and rename the node u as v.

Now consider the insertion. Let p be a point to be inserted. The insertion algorithm has two steps. First we find the smallest box containing the point p. Then we update a finite set of nodes and boxes of the tree T. The first step uses the point location algorithm that is described in Section 5. After point location there are 3 cases.

1. The point p does not belong to $B(v_{\texttt{root}})$, where $v_{\texttt{root}}$ is the root of T.

2. The point p belongs to the box $B(v)$, where v is a leaf (see Fig. 2 a)).

3. The point p belongs to the set $B(v) \setminus SB(v)$ for some node v (see Fig. 2 b)).

The cases 1 and 2 can be handled similarly to case 3. Consider the case 3. We want to construct an s-box D and a fair split of D into the boxes D_1 and D_2 that satisfy the following conditions

- the box D is an s-sub-box of $B(v)$,
- the box $SB(v)$ is an s-sub-box of D_1, and
- the point $p \in D_2$.

After finding D, we remove the edges from v to children v' and v'', create two nodes u and w below v, add edges joining u to v' and v'', and set $SB(u) = SB(v), B(u) = D_1, SB(v) = D, B(w) = D_2, SB(w) = D_2$ (see Fig. 2 b)).

Denote $SB(v) = [a_1, b_1) \times \ldots \times [a_k, b_k)$. The algorithm uses a box D and repeatedly shrinks the box D until a fair split of D is found. Initially $D = B(v)$. Denote $D = [d_1, e_1) \times \ldots \times [d_k, e_k)$. After each iteration of the algorithm

1) the box D is an s-box and an s-sub-box of $B(v)$,

2) the box $SB(v)$ is an s-sub-box of D, and

3) the box D contains the point p.

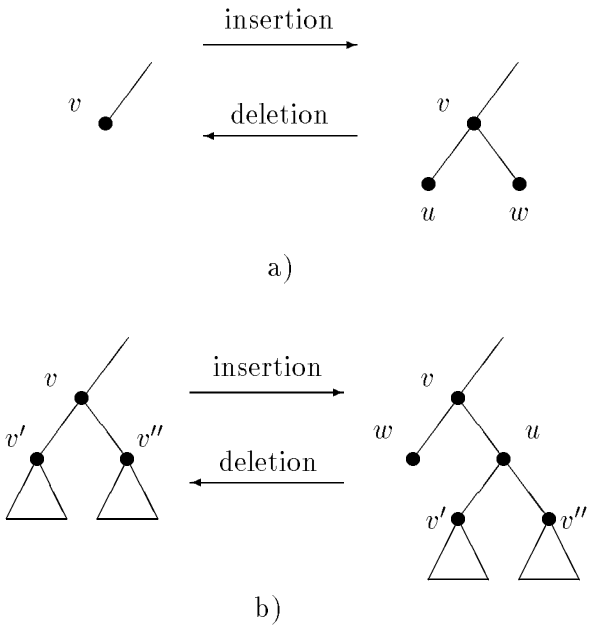

Figure 2: Updating the fair-split tree. a) The inserted point p belongs to $B(v)$. The deleted point p belongs to $B(u)$ where u is a son of v. b) The inserted point p belongs to $B(v) \setminus SB(v)$. The deleted point p belongs to $B(w)$.

The algorithm has $O(1)$ iterations because after each iteration the number of coordinates a_i, b_i coinciding with endpoints of $[d_i, e_i)$ is increased, i.e. the sum $\sum_{i=1}^{k} |\{a_i, b_i\} \cap \{d_i, e_i\}|$ is increased. We shall call this the number of connected endpoints. The basic procedure is the *fair-split* procedure.

procedure *fair-split* (D) $(*$ fair split of the box $D = [d_1, e_1) \times \ldots \times [d_k, e_k)$ $*)$

1) Find i such that $e_i - d_i$ is maximal. In Step 2 we choose $const \in [d_i, e_i)$ to partition D by the hyperplane $x_i = const$. Compute the interval $[d_i', e_i'] = [d_i + \frac{e_i - d_i}{s+1}, e_i - \frac{e_i - d_i}{s+1}]$ which contains all possible values of *const*.

2) If a_i or b_i lies in the interval $[d_i', e_i']$, then $const = a_i$ or $const = b_i$, respectively. Otherwise the interval $[a_i, b_i]$ does not intersect the interval $[d_i', e_i']$. There are two possible cases.

2.1) $b_i < d_i'$ (in other words, $[a_i, b_i) \subseteq [d_i, d_i')$). Let $d_i'' = b_i + \frac{b_i - d_i}{s}$ (d_i'' is minimal real number such that the split of $[d_i, d_i'']$ by b_i is fair). Then $const = \max(d_i', d_i'')$.

2.2) $a_i > e_i'$ (in other words, $[a_i, b_i) \subseteq [e_i', e_i)$). Let $e_i'' = a_i - \frac{e_i - a_i}{s}$ (e_i'' is maximal real number such that the split of $[e_i'', e_i]$ by a_i is fair). Then $const = \min(e_i', e_i'')$.

3) Partition the box D by the hyperplane $x_i = const$. If this hyperplane separates the box $SB(v)$ and the point p, the cut of D into the boxes $D \cap \{x, x_i < const\}$ and $D \cap \{x, x_i \geq const\}$ is a fair split which satisfies the above conditions (1), (2), and (3). In this case we stop the iteration. Otherwise one of these boxes contains both the box $SB(v)$ and the point p. Choose this box as D.

4) End of procedure.

Now we shall describe the iteration of the algorithm. If, for some j, the interval $[\min(a_j, p_j), \max(b_j, p_j)]$ intersects the interval $[d_j', e_j')$, then call the fair-split procedure until the number of connected endpoints increases ($const = a_j$ or $const = b_j$ in Step 2) or the iteration finishes (in Step 3). The procedure *fair-split* splits i-th side of D at most $O(1)$

times (more precisely, 3 times for $s = 2$ and 2 times for $= \frac{\sqrt{5}+1}{2}$). The number of calls is at most $O(k)$.

For any j, the interval $[\min(a_j, p_j), \max(b_j, p_j)]$ does not intersect the interval $[d'_j, e'_j]$. Without loss of generality, $b_j < d'_j$ for all j. Choose j such that $c_j = \frac{\max(b_j,p_j)-d_j}{e_j-d_j}$ is maximal. The box $[d_1, d_1 + \frac{s+1}{s}c_j(e_1 - d_1)) \times \ldots \times [d_k, d_k + \frac{s+1}{s}c_j(e_k - d_k))$ is an s-box and s-sub-box of $B(v)$. Shrink D to this box. Then $[\min(a_j, p_j), \max(b_j, p_j)]$ intersects the middle interval of $[d_j, e_j]$ and we obtain the preceding case.

Hence we have proved the following result.

Theorem 3.1 *Let the dimension k be fixed and let point location take $COST$ time. A fair-split tree T can be maintained in $O(1) + COST$ time per insertion and $O(1)$ time per deletion.*

4 The maintenance of the closest pair

To maintain the closest pair we shall store the set E of some pairs of points of S.

Definition 4.1 A point $p \in S$ is a *nearest neighbor of* q if, for any $r \in S \setminus \{q\}$, $d(p,q) \leq d(q,r)$. For points $p, q \in S$, we call the pair (p,q) a *neighbor pair* if p is the nearest neighbor of q and vice versa.

The set E contains the neighbor pairs. It is clear that the closest pair of S is a neighbor pair of S and the closest pair belongs to E.

Let a heap H store the distances of the pairs of E. The heap item is the pair of the points. The key of the item (p,q) is the L_t-distance $d(p,q)$. The pair of points with minimal key is a closest pair of S.

With each point $p \in S$, we store a list $E_p = \{q \mid (p,q) \in E\}$. With each point q in E_p, we store a pointer to the item (p,q) of the heap H.

Definition 4.2 An ordered pair [1] (a,b) of points from S is an *ordered rejected pair* if there exists a node v in the fair-split tree satisfying the following:

1. $a \notin B(v)$
2. $d(B(v)) \leq sd(B(a))$
3. $d_{\min}(a, B(v)) \leq (1+s)d(B(v))$
4. $d_{\max}(a, B(v)) < d(a,b)$.

An unordered pair (a,b) of points from S is a *rejected pair* if ordered pair (a,b) or (b,a) is an ordered rejected pair.

The set E satisfies the following property.

Invariant. For any distinct points $a, b \in S$, the unordered pair (a,b) belongs to the set E unless (a,b) is a rejected pair.

Lemma 4.3 *Let the invariant hold for the set E. Then the set E contains the neighbor pairs of S.*

Proof. By the condition 4 of Definition 4.2.□

It is easy to see that the set of all pairs satisfy the invariant. We maintain the additional invariant that, for any $p \in S$, the number of incident pairs in E is at most constant, i.e. $|E_p| = O(1)$. This gives us a linear bound on $|E|$. We can bound $|E_p|$ by the following statement.

Statement 4.4 *For any point $p \in S$, the number of non-rejected pairs $(p,q) \in S$ is at most $O(1)$.*

[1] We shall define by (a,b) either an unordered pair $\{a,b\}$ or an ordered pair $[a,b]$, using the context to resolve the ambiguity.

Let $N_k = (24k+1)^k$. We shall prove that the number of non-rejected pairs incident to a point p is at most N_k (for the separation $s = 2$). It is important that this bound is independent of n.

Statement 4.4 follows from Theorem 4.6. We precede Theorem 4.6 with useful Lemma.

Lemma 4.5 *Let p and q be points of S. If $d(p,q) > (1+s)d(B(p))$ then the pair (p,q) is rejected.*

Proof. Consider the leaf u corresponding to the point p. Let v be the sibling of u and $a = p$. The point a and the node v satisfy the conditions 1, 2, and 3 of Definition 4.2. The pair (p,q) is rejected if $d(p,q) > d_{\max}(p, B(v))$. Lemma follows from $d_{\max}(p, B(v)) \leq d_{\max}(B(u), B(v)) \leq d(B(u)) + d(B(v)) \leq (1+s)d(B(u))$. □

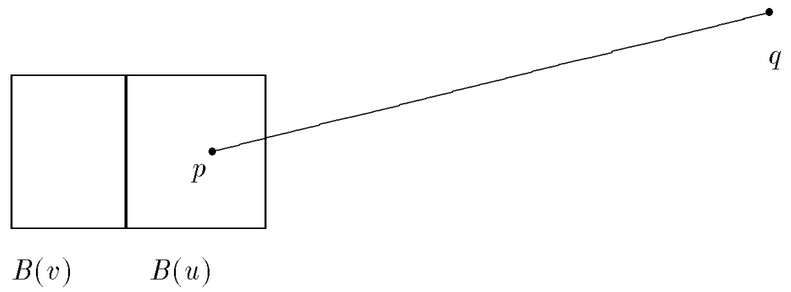

Figure 3: The distance between points p and q is greater than the diameter of the box $B(u)$ times $s+1$. The pair (p,q) is rejected.

Theorem 4.6 is useful in the insertion algorithm. To find a set E_p, for an inserted point p, we use a search on the dynamic tree. We need to limit the number of nodes that are used in search at the same time. Let $V = \{v_1, \ldots, v_N\}$ be a set of these nodes. We associate the set $S_i = B(v_i) \setminus \bigcup_{B(v_j) \subset B(v_i)} B(v_j)$ with every node $v_i \in V$.

Theorem 4.6 *Let p be a point of S, $V = \{v_1, \ldots, v_N\}$ be a set of nodes of a fair-split tree T. If $N > N_k$, there exists i such that, for any $q \in S_i \cap S$, the pair (p,q) is rejected. (Choosing of i does not depend on layout of the points of S in the associated sets).*

Proof. We can assume that, for any i, the intersection $S_i \cap S \neq \emptyset$ and there exists a point $q \in S_i \cap S$ such that the pair (p,q) is non-rejected. (In fact we can recognize whether an empty set $S_i \cap S$ exists in $O(N)$ time. For an index j, the set $S_i \cap S$ is empty if and only if any leaf below the node v_j has an ancestor which is a descendant of v_j.)

Choose a box $B(v_i)$ of minimum diameter. Let $\delta = d(B(v_i))$. First we shall prove that, for any point q at distance greater than $(2+s)\delta$ from p, the pair (p,q) is rejected. We consider three cases.

Case 1. The point p belongs to the box $B(v_i)$.
Then $B(p) \subseteq B(v_i)$ and $\delta \geq d(B(p))$. For any point q with $d(p,q) > (1+s)\delta$ the pair (p,q) is rejected by Lemma 4.5. Hence we can assume

$$p \notin B(v_i). \tag{1}$$

Case 2. The diameter of $B(p)$ is less than δ/s.
Recall that the separator s is greater than 1. Let q be any point at distance greater than $(2+s)\delta$ from p. Then, since $s > 1$, $d(p,q) > (1+s)\delta/s > (1+s)d(B(p))$ and the pair (p,q)

is rejected by Lemma 4.5. Hence we can assume

$$\delta \leq sd(B(p)). \tag{2}$$

Case 3. The distance from the point p to the box $B(v_i)$ is greater than $(1+s)\delta$. Choose any point q from $S \cap B(v_i)$. It is clear that $B(q) \subseteq B(v_i)$ and $d(p,q) > (1+s)d(B(q))$. The pair (p,q) is rejected by Lemma 4.5 and the node v_i can be removed from V. This contradicts our assumptions. Hence we can assume

$$d_{\min}(p, B(v_i)) \leq (1+s)\delta. \tag{3}$$

Let $a = p$ and $v = v_i$. Choose any point $b \in S$ such that $d(a,b) > (2+s)\delta$. The conditions 1, 2, and 3 of Definition 4.2 are the assumptions (1),(2), and (3). Note that $d(a,b) > \delta + d_{\min}(p, B(v_i)) \geq d_{\max}(a, B(v))$. Hence the pair (a,b) can be removed from E.

Thus, we can remove a node v_j from V if $d_{\min}(p, S_j) > (2+s)\delta$. The number of nodes v_j such that $d_{\min}(p, S_j) \leq (2+s)\delta$ is at most $N_k = (24k+1)^k$ by Lemma 4.7. The result follows. □

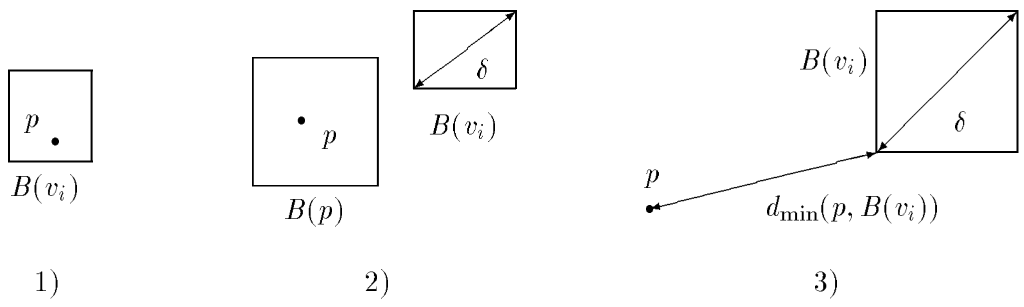

Figure 4: Three cases of Theorem 4.6. (1) $p \in B(v_i)$. (2) $d(B(p)) < \delta/s$. (3) $d_{\min}(p, B(v_i)) > (1+s)\delta$.

Lemma 4.7 *Let p be a point of S, $V = \{v_1, \ldots, v_N\}$ be a set of nodes of a fair-split tree T such that, for any j, the set $S_j = B(v_j) \setminus \bigcup_{B(v_i) \subset B(v_j)} B(v_i)$ is nonempty. Let δ be a minimum diameter $d(B(v_i))$, $v_i \in V$. If, for any j, the distance $d_{\min}(p, S_j) \leq (2+s)\delta$, then the number $N \leq N_k = (24k+1)^k$.*

Proof. Fix any $j \in \{1, \ldots, N\}$. Choose the point $q \in S_j$ such that $d(p,q) \leq (2+s)\delta$. Note that the box $B(v_{\mathtt{root}})$ corresponding to the root of T contains the point q and any box $B(v)$, $v \in V$. We shall show that q is included in S_j together with some s-box C, any side of C is at least $\delta/((1+s)k)$. If $S_j = B(v_j)$ then $C = B(v_j)$. Otherwise choose the minimal box $B(u)$, $u \in T$ that contains the point q and at least one box $B(v)$ for some $v \in V$.

We distinguish two cases. In the first case, the point q belongs to the box $SB(u)$. Note that $B(v) \subseteq SB(u)$. Then the fair split of the box $SB(u)$ separates the point q and the box $B(v)$, i.e. the point $q \in B(u_1)$, the box $B(v) \subseteq B(u_2)$ where u_1, u_2 are the sons of u. Note that

- $d(B(u_2)) \geq \delta$,
- the length of the longest side of $B(u_2)$ is at least δ/k, and
- the length of the shortest side of $B(u_2)$ is at least $\delta/((1+s)k)$.

The sides of box $B(u_1)$ are equal to the sides of box $B(u_2)$ except for those that are a part of the partitioned side of box $B(u)$. Now consider the second part of this side. If it is a longest side of $B(u_2)$ then the corresponding side of $B(u_1)$ has length at least $\delta/((1+s)k)$. Otherwise one of the sides of $B(u_1)$ has length at least δ/k. Hence any side of $B(u_1)$ has length at least $\delta/((1+s)k)$. In the first case the point q is included in S_j together with the box $B(u_1)$ and any side of $B(u_1)$ is at least $\delta/((1+s)k)$.

In the second case, the point q does not belong to the box $SB(u)$, i.e. the point $q \in B(u) \setminus SB(u)$. Note that $B(v) \subseteq SB(u)$. This situation is similar to the case 3 of the insertion algorithm (Section 3). We proved that there exists an s-box D and a fair split of D into boxes D_1 and D_2 such that

- the box D is an s-sub-box of $B(u)$,
- the box $SB(u)$ is an s-sub-box of D_1, and
- the point $q \in D_2$.

The diameter of the box D_1 is at least δ. The situation is similar to the first case and we can show that any side of D_2 is at least $\delta/((1+s)k)$.

Thus, in both cases, there exists an s-box $C \subseteq S_j$ that contains the point q, and any side of C is at least $\delta/((1+s)k)$. The box C contains at least one point of the lattice

$$L = \{x \mid \frac{x_i - p_i}{\sigma} \in Z, \text{ where } \sigma = \frac{\delta}{(1+s)k}, \text{ and } i = 1, \dots, k\}.$$

Let r be a point of $C \cap L$ closest to p. Then, for any i

$$|r_i - p_i| \leq \left\lceil \frac{(2+s)\delta}{\sigma} \right\rceil \sigma = \lceil (2+s)(1+s)k \rceil \sigma.$$

Therefore the set S_j contains at least one point among the points in the set

$$L' = \{x \mid \frac{x_i - p_i}{\sigma} \in \{-l_k, \dots, 0, \dots, l_k\}, \text{ where } l_k = \lceil (2+s)(1+s)k \rceil \text{ for } i = 1, \dots, k\}.$$

For the separation $s = 2$ the cardinality of this set is $N_k = (24k+1)^k$. This implies that $N = |V| \leq N_k$. □

The insertion algorithm uses Theorem 4.6 if the set V contains more than N_k nodes. We describe the algorithm to refine the set V (in Section 7 we give effective algorithms to refine node sets in searching E_p and $A(v)$).

Algorithm *REFINE(V)* (∗ this algorithm is used in searching for E_p ∗)

1. Remove the nodes $v_j \in V$ such that

$$d_{\min}(p, B(v_j)) > (1+s)d(B(v_j)) \text{ or } d_{\min}(p, S_j) > (1+s)d(B(p))$$

2. Compute $\delta = \min_{v_j \in V}\{d(B(v_j))\}$.
3. Remove the nodes $v_j \in V$ such that

$$d_{\min}(p, S_j) > (2+s)\delta$$

The insertion of the point p causes insertion of some pairs into E and deletion of some pairs from E. Let us look at the updates of boxes. Note that the boxes, corresponding to the nodes, are only inserted and, in the case $B(v_{\texttt{root}})$, are enlarged. Hence to prove that the invariant holds for E we need not insert pairs that are not incident to an inserted point. Using the dynamic tree we find at most N_k pairs that are adjacent to p. Add these pairs into E. Now in fact the invariant holds for E. However, for some points, the number of incident pairs may exceed N_k. These points are adjacent to p and can be determined when adding pairs into E. For these points, we remove some pairs from E using Theorem 4.6.

Now we consider the deletion of the point p. The deletion causes insertion of some pairs into E and deletion of some pairs from E. Delete the pairs adjacent to p, i.e. the set $\{(p,q) \mid q \in S, (p,q) \in E\}$. Note that always two boxes are deleted. These boxes are the results of a fair split of the box $SB(parent(w))$ where the node w corresponds to p.

We consider the deletion of the box $B(v)$. Suppose that the pair (a,b) was rejected (and was not included in E) by conditions of Definition 4.2 for node v. Then $d(B(a)) \geq d(B(v))/s$ and $d_{\min}(a, B(v)) \leq (1+s)d(B(v))$. We shall show that the number of such points is at most $O(1)$. The argument is similar to the proof of Theorem 4.6. Let $A(v)$ denote this set, i.e.

$$A(v) = \{a \in S \mid d(B(a)) \geq d(B(v))/s \text{ and } d_{\min}(a, B(v)) \leq (1+s)d(B(v))\}.$$

For each $a \in A(v)$, we renew the set E_a. This gives the set E, for which the invariant is fulfilled (if we renew the sets for both deleted boxes). For the points $q \in S$, $|E_q| > N_k$, remove some points from E_q using Theorem 4.6. Now the second invariant ($|E_q| \leq N_k$, for any $q \in S$) holds.

In the rest of this Section we prove the analog of Theorem 4.6 for $A(v)$. Denote $M_k = (36k+19)^k$. To find a set $A(v)$ we use a search on the dynamic tree. As in finding of E_p we bound the number of nodes that are used in search at the same time. We shall prove that this number is at most M_k (for the separation $s = 2$). Let $V = \{v_1, \ldots, v_N\}$ be a set of these nodes. We associate the set $S_i = B(v_i) \setminus \bigcup_{B(v_j) \subseteq B(v_i)} B(v_j)$ with every node $v_i \in V$.

Theorem 4.8 *Let v be a node of a fair-split tree T, $V = \{v_1, \ldots, v_N\}$ be a set of nodes of T. If $N > M_k$, there exists i such that $A(v) \cap S_i = \emptyset$ (choosing of i does not depend on layout of the points of S in the associated sets).*

Proof. We can assume that, for any i, $S_i \cap S \neq \emptyset$. Let δ be a minimum diameter $d(B(v_i))$, for $v_i \in V$. Note that $\delta \geq d(B(v))/s$. By definition of $A(v)$ we can assume that, for any j,

$$d_{\min}(S_j, B(v)) \leq (1+s)d(B(v)).$$

Fix any $j \in \{1, \ldots, N\}$. Choose the point $q \in S_j$ such that $d_{\min}(q, B(v)) \leq (1+s)d(B(v))$. As in the proof of Theorem 4.6 we can show that there exists s-box C satisfying the following:

- $q \in C$,
- any side of C has length at least $\delta/((1+s)k)$.

The box C contains at least one point of the lattice

$$L = \{x \mid \frac{x_i - p_i}{\sigma} \in Z, \text{ where } \sigma = \frac{\delta}{(1+s)k}, \text{ and } i = 1, \ldots, k\}.$$

Let p be the center of the box $B(v)$. The shortest side of $B(v)$ has length at least $d(B(v))/k$. The longest side of $B(v)$ has length at least $(1+s)d(B(v))/k$. Hence

$$|q_j - p_j| \leq \frac{(1+s)d(B(v))}{2k} + (1+s)d(B(v)) \leq s(1+s)(1+\frac{1}{2k})\delta.$$

Let r be a point of $C \cap L$ closest to p. Then, for any i

$$|r_i - p_i|/\sigma \leq \lceil s(1+s)(1+\frac{1}{2k})\delta/\sigma \rceil = \lceil s(1+s)^2(k+\frac{1}{2}) \rceil.$$

Therefore the set S_j contains at least one point among the points in the set

$$L' = \{x \mid \frac{x_i - p_i}{\sigma} \in \{-l_k, \ldots, 0, \ldots, l_k\}, \text{ where } l_k = \lceil s(1+s)^2(k+\frac{1}{2}) \rceil \text{ for } i = 1, \ldots, k\}.$$

For the separation $s = 2$ the cardinality of this set is $M_k = (36k+19)^k$. This implies that $N = |V| \leq M_k$. □

5 Dynamic tree

In this Section we shall briefly describe the dynamic tree. We use the dynamic tree to implement the point location and other searches on the fair-split tree.

A dynamic tree $\Delta(T)$, based on the binary tree T, has the same nodes and the same edges as T. The dynamic tree is a partition of edges into two kinds, *solid* and *dashed*, with property that each node has at most one child linked to it by a solid edge. Thus the solid edges define a collection of *solid paths* that partition the vertices. (A vertex with no incident solid edges is a one-vertex solid path). The head of the path is its bottommost node; the tail is its topmost node.

For a node v of T, let $size(v)$ be the number of nodes in the subtree of T rooted at v. Let (v, w) be an edge of T from v to its parent w. The edge is *heavy* if $size(v) > size(w)/2$ and *light* otherwise. A node v of $\Delta(T)$ fulfills the *size invariant* if, for each edge e to one of its children, e is solid if it is heavy and light if it is dashed. We say that the size invariant holds for the dynamic tree $\Delta(T)$ if it holds for each node of T.

A solid path is represented by a *path tree*. We use *globally biased binary trees* [4] to implement path trees. A biased binary tree stores an ordered sequence of *weighted* items in its leaves. The weight of a node v of T (and of the corresponding leaf of the biased binary tree) is defined as

$$weight(v) = \begin{cases} size(v), \text{ if no solid edge enters } v \\ size(v) - size(w), \text{ if the solid edge } (w, v) \text{ enters } v \end{cases}$$

The weight of an internal node of a biased binary tree is inductively defined as the sum of the weight of its children.

Each node v of biased binary tree has an integer rank denoted $rank(v)$ that satisfies the following properties:

(i) If v is a leaf, $rank(v) = \lfloor \log weight(v) \rfloor$. If v is an internal node, $rank(v) \leq 1 + \lfloor \log weight(v) \rfloor$.

(ii) If node w has parent v, $rank(w) \leq rank(v)$, with the inequality strict if w is external. If w has grandparent u, $rank(w) < rank(u)$.

Each internal node v of biased binary tree contains four pointers [27]: $bleft(v)$ and $bright(v)$, which point to the left and right child of v, and $bhead(v)$ and $btail(v)$, which point to the head and tail of the subpath corresponding to v (the leftmost and rightmost external descendants of v). For a topmost node v of a solid path P, there is the pointer $pt\_root(v)$ to the root of the path tree for P.

Lemma 5.1 ([27]) *If v is a leaf of a biased binary tree with root u, the depth of v is at most $2(rank(u) - rank(v)) \leq 2\log(weight(u)/weight(v)) + 4$.*

The updates of T can be performed using the following operations [4] on rooted trees.

$link(v, w)$: If v is the root of one tree and w is a node in another tree, combine the trees containing v and w by adding an edge joining v and w.

$cut(v, w)$: If there is an edge joining v and w, delete it, thereby breaking the tree containing v and w into two trees, one containing v and one containing w.

The time bound of these operations is $O(\log n)$. This gives the following result.

Lemma 5.2 *The dynamic tree can be maintained under insertions and deletions of points in $O(\log n)$ time per update.*

6 Searching

In this Section we discuss the search algorithms. We have to implement point location and the search for the sets E_p and $A(v)$.

6.1 Point location

Let p be a point in k-dimensional space. The nodes of T whose boxes contain p form the path (if $p \in B(v_{\mathtt{root}})$). We have to compute the bottommost node of this path. Our point location algorithm is similar to the algorithm of Schwarz [24]. The algorithm processes a sequence of solid paths of the dynamic tree. For any solid path P of this sequence, the box of the topmost node of P contains p.

We start the algorithm with the solid path containing the root. If the box $B(v_{\mathtt{root}})$ does not contain p then the algorithm returns **null**.

Now assume that the algorithm has reached the topmost node of the solid path P, and p is contained in the box of that node. We find the lowest node v on P whose box still contains the query point p. At this point we continue the search with a dashed edge (v, u) such that $p \in B(u)$. It is clear that the node u is the topmost node of the next solid path.

Now we describe the search on the solid path P. The algorithm start with the root u of the path tree. We execute the following step until u is a leaf of the path tree. Follow the pointer from u to the rightmost leaf in the u's left subtree. This node is $btail(bleft(u))$. If the box $B(btail(bleft(u)))$ contains the query point, then we proceed with u's left child in the path tree, otherwise with the right child.

function point_location(p)
  $v := root(T)$
  if $p \notin B(v)$ **then return null**
  while v is an internal node of T **do**
    ($*$ Note that $p \in B(v)$ and v is the topmost node of some path P $*$)
    $u := pt\_root(v)$ ($*$ u is the root of the path tree for P $*$)
    while u is an internal node of the path tree **do**
      if $p \in B(btail(bleft(u)))$ **then**
        $u := bleft(u)$
      else $u := bright(u)$
      fi
    od
    ($*$ u is the bottommost node of the path P such that $p \in B(u)$ $*$)
    $v := u$
    if the edge $(v, rson(v))$ is dashed and $p \in B(rson(v))$ **then**
      $v := rson(v)$
    else if the edge $(v, lson(v))$ is dashed and $p \in B(lson(v))$ **then**
          $v := lson(v)$
        else return v
        fi
    fi
  od
  return v
end ($*$ of the function $*$)

It is clear that the point location algorithm is correct. Let us analyze the running time of the algorithm. Let $P_1, \ldots, P_l$ be the solid paths that are searched during the algorithm. Let

$u_1, \ldots, u_l$ be the roots of path trees and $v_1, \ldots, v_l$ be the bottommost nodes on path trees that are searched. Note that v_i is the parent of u_{i+1} in T for $i = 1, \ldots, l-1$. The number l of paths is at most $\log n$ by the size invariant. The depth of v_i in the path tree for P_i is at most $2(rank(u_i) - rank(v_i))$ by Lemma 5.1. For $i = 1, \ldots, l-1$, $rank(v_i) \geq rank(u_{i+1})$ by definition of $rank$. The total running time of the point location algorithm is

$O(\log n + \sum_{i=1}^{l} 2(rank(u_i) - rank(v_i))) = O(\log n + rank(u_1) - rank(v_l)) = O(\log n)$.

6.2 Searching for E_p and $A(v)$

Now we shall describe the search for the sets E_p and $A(v)$. Recall that $E_p = \{q \mid (p, q) \in E\}$ and

$$A(v) = \{a \in S \mid d(B(a)) \geq d(B(v))/s \text{ and } d_{\min}(a, B(v)) \leq (1+s)d(B(v))\}.$$

We consider the search for E_p and $A(v)$ as a point location problem for at most $O(1)$ points (N_k points for E_p and M_k points for $A(v)$). In fact we can build a search tree such that

- the external nodes correspond the points S, and
- the path from the root of the search tree to an external node v corresponds to the nodes of the path trees searched during the location of the point corresponding to v.

The search for the sets E_p and $A(v)$ applies *breadth-first search* on the search tree. *node_set* denotes a set of nodes that is stored in the breadth-first search. We use the pointer $depth(v)$ that is a depth of the node v in search tree. For simplicity, we extend the pointers $btail$ to the external nodes of any path trees. (It is not necessary to store these pointers). Using Theorem 4.6 (4.8), the procedure *refine()* finds at most N_k (resp.M_k) nodes among the nodes $\{btail(v) \mid v \in node\_set\}$ and removes another nodes from *node_set*.

```
function search() (* the search for E_p or A(v) *)
  w := pt_root(root(T))
  node_set := {w}
  depth(w) := 0
  while there is a node w in node_set such that btail(w) is an internal node of T do
    w is a node in node_set with minimal depth such that btail(w) is an internal node of T
    if w is an internal node of some path tree then
      node_set := node_set ∪ {bleft(w), bright(w)}
      depth(bleft(w)) := depth(w) + 1
      depth(bright(w)) := depth(w) + 1
    else (* w is an external node of some path tree *)
      u := btail(w) (* u is the corresponding node of w in T *)
      if the edge (u, rson(u)) is dashed then
        w := pt_root(rson(u))
        node_set := node_set ∪ {w}
        depth(w) := depth(w) + 1
      fi
      if the edge (u, lson(u)) is dashed then
        w := pt_root(lson(u))
        node_set := node_set ∪ {w}
        depth(w) := depth(w) + 1
      fi
    fi
    node_set := node_set \ {w}
```

if $|node\_set| > N_k$ **then** (* $|node\_set| > M_k$ for $A(v)$ *)
refine($\{btail(w) \mid w \in node\_set\}$)
(* by Theorem 4.6 for E_p and Theorem 4.8 for $A(v)$ *)
fi
od
return the points corresponding the nodes $btail(w)$ for $w \in node\_set$
end (* of the function *)

Lemma 6.1 *The function search() takes $O(\log n)$ time.*

Proof. The function search() visits at most N_k (resp.M_k) nodes of the same depth. The depth of the search tree is $O(\log n)$. This completes the proof. □

Finally, we formulate the main result.

Theorem 6.2 *There is a data structure of size $O(n)$ that maintains the closest pair of S in $O(\log n)$ time per update.*

7 The reduction of the constant factors

In this Section we discuss the dependence of the update time and the space on dimension. The complexity of the algorithm is exponential in the dimension. The straightforward implementation of the searching gives $O(kN_k^2 \log n)$ time to insert and $O(kM_k(M_k + N_k^2) \log n)$ time to delete a point. This is because the procedure *refine()* takes $O(kN_k)$ time in the searching for E_p and $O(kM_k)$ time in the searching for $A(v)$.

Now we shall reduce the time complexity of *refine()* to $O(k + \log N_k)$ and $O(k + \log M_k)$ respectively. Instead of computing the minimum diameter box $B(v_i)$ (in $O(N_k)$ time), we shall maintain it. Note that the node v_i is never deleted. In the loop of *search*() we have to choose a node v such that $btail(v)$ is an internal node of T. To do this we store *node_set* in two lists: $\{v \mid btail(v)$ is an internal node of $T\}$ and $\{v \mid btail(v)$ is an external node of $T\}$. Using the queue for the first list allows us to find a node with minimal depth in $O(1)$ time.

Consider the search for E_p. We can formulate the conditions to remove the node v_j

$$d_{\min}(p, B(v_j)) > (1 + s)d(B(v_j)) \tag{4}$$

$$d_{\min}(p, S_j) > (1 + s)d(B(p)) \tag{5}$$

$$d_{\min}(p, S_j) > d_{\max}(p, B(v_i)) \tag{6}$$

In fact we check these conditions when we add a node to *node_set*.

Consider the search for $A(v)$. The following conditions allows us to discard inserted node v_j

$$d(B(v_j)) > d(B(v))/s \tag{7}$$

$$d_{\min}(S_j, B(v)) > (1 + s)d(B(v)) \tag{8}$$

The conditions 4, 5 and 7 can be computed in $O(k)$ time. We can achieve the same time bound for the conditions 6 and 8. The main problem is how to compute S_j. Recall that $S_j = B(v_j) \setminus \bigcup_{B(v_i) \subset B(v_j)} B(v_i)$ for a node $v_j \in node\_set$. Instead of computing this set, we compute its subset such that Theorems 4.6 and 4.8 still hold.

Let w be a node of some path tree and w is added to *node_set* ($v_j = btail(w)$). Let $q \in \mathbf{R}^k$ be a point such that the point location of p visits w. It is clear that $q \in S_j$. In fact we can take the set of such points to be S_j. In other words, we can define

$$S_j = \begin{cases} B(btail(w)) \setminus B(btail(lson(u)), \text{if } w \text{ is right son of } u \\ B(btail(w)), \text{otherwise} \end{cases}$$

The set S_j is either a box or the set theoretical difference between two boxes. This definition of set S_j is similar to the definition of *cells* [1] (*box cells* and *doughnut cells*). The conditions 6 and 8 can be computed in $O(k)$ time.

In practice, we don't need to store the at most $|N_k|$ ($|M_k|$ for $A(v)$) nodes in *node_set*. We can prune *node_set* at the moment we add a node to *node_set*. To do this we store $d_{\min}(p, S_j)$ ($d_{\min}(S_j, B(v))$ for $A(v)$) in a heap corresponding to *node_set*. Then the cost of insertion a node to *node_set* is $O(k + \log N_k) = O(k \log k)$ ($O(k + \log M_k) = O(k \log k)$ for $A(v)$). The deletion of a node from *node_set* take $O(k + \log N_k) = O(k \log k)$ ($O(k + \log M_k) = O(k \log k)$ for $A(v)$) time. Hence the search for E_p (for $A(v)$) takes $O(kN_k \log k \log n)$ (resp. $O(kM_k \log k \log n)$) time.

Now consider the insertion of the point p. Recall that after finding E_p we have to prune the sets $E_q, q \in E_p$ containing greater than N_k points. We can prune a set E_q in $O(k + \log N_k)$ time. We shall store two heaps to node q. The keys are the distances $d(B(r))$ and $d_{\min}(q, B(r))$, $r \in E_q$ (for these points $S_j = B(r)$). The total time of insertion the point p is $O(kN_k \log k \log n + N_k(k + \log N_k)) = O(kN_k \log k \log n)$.

We now consider the deletion of the node v. Recall that after finding $A(v)$, for each $a \in A(v)$, we

- delete the set E_a,
- find the set E_a, using the search for E_p,
- prune $E_b, b \in E_a$, if $|E_b| > N_k$,

The corresponding costs are $O(M_k N_k \log N_k)$, $O(kM_k N_k \log N_k \log n)$ and $O(M_k N_k \log N_k)$. The total running time of the deletion algorithm is $O(kM_k N_k \log k \log n)$.

Theorem 7.1 *There is a data structure of size $O(kn)$ that maintains the closest pair of S in $O(kN_k \log k \log n)$ time per insertion and $O(kM_k N_k \log k \log n)$ time per deletion.*

Finally, we compare constants N_2 and M_2 for separation $s = 2$ and $s = \frac{\sqrt{5}+1}{2}$. Recall that $N_k = (2\lceil (s+2)(s+1)k \rceil + 1)^k$ and $M_k = (\lceil s(s+1)^2(k+\frac{1}{2}) \rceil + 1)^k$. For separation $s = 2$ we get $N_2 = 2401$ and $M_2 = 8281$. For separation $s = \frac{\sqrt{5}+1}{2}$ we get $N_k = (2\lceil 9.4721k \rceil + 1)^k$, $N_2 = 1521$ and $M_k = (2\lceil 11.0901k + 5.5450 \rceil + 1)^k$, $M_2 = 3249$. In practice, we do not expect the constant factors to be so big.

8 Conclusion

We have presented an algorithm for maintaining the closest pair in $O(\log n)$ time per update, using $O(n)$ space. The running time of the algorithm is optimal up to a constant factor in the algebraic decision-tree model of computation. The algorithm can be adapted (by changing some constants, including N_k) for another metric such that $d(p, q) = O(d_\infty(p, q))$. In fact the algorithm can give the list of the closest pairs (if any) in the time proportional to its number.

The algorithm maintains a set E of point pairs that contains the neighbor pairs.

Unfortunately the fair-split tree does not allow efficiently maintaining the (exact) set of the neighbor pairs. It would be interesting to solve the problem of the neighbor pairs maintenance with $O(\log n)$ update time and $O(n)$ space.

Acknowledgment. The author thanks anonymous referees for many useful comments.

References

[1] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest-neighbor searching. Proceedings 5th Annual Symposium on Discrete Algorithms, 1994, pp. 573–582.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest-neighbor searching. (revised version), 1994.

[3] M. Ben-Or. Lower bounds for algebraic computation trees. Proc. 15th Annual ACM Symp. Theory Comput., 1983, pp. 80-86.

[4] S. W. Bent, D. D. Sleator and R.E. Tarjan Biased search trees. SIAM Journal of Computing, 1985, 14, pp. 545-568.

[5] J. L. Bentley and M. I. Shamos. Divide-and-conquer in multidimensional space. Proc. 8th Annual ACM Symp. Theory of Computing, 1976, pp. 220-230.

[6] S. N. Bespamyatnikh. The region approach for some dynamic closest-point problems. Proc. 6th Canadian Conf. Comput. Geom., 1994, pp. 75–80.

[7] S. N. Bespamyatnikh. An optimal algorithm for closest pair maintenance. Proc. 11th Annual ACM Symp. Comput. Geom., 1995, pp. 152–161.

[8] S. N. Bespamyatnikh. An optimal algorithm for dynamic post-office problem in R_1^2 and related problems. Proc. 8th Canadian Conf. Comput. Geom., 1996, pp. 101–106.

[9] S. N. Bespamyatnikh. Dynamic algorithms for approximate neighbor searching. Proc. 8th Canadian Conf. Comput. Geom., 1996, pp. 252–257.

[10] P. B. Callahan. Dealing with Higher Dimensions: The Well-Separated Pair Decomposition and Its Applications. Ph.D. Thesis, The Johns Hopkins University, 1995.

[11] P. B. Callahan, M. T. Goodrich and K. Ramaiyer. Topology B-trees and their applications. Proc. 4th Workshop Algorithms Data Struct., 1995, pp. 381–392.

[12] P. B. Callahan and S. R. Kosaraju. A decomposition of multi-dimensional point-sets with applications to k-nearest-neighbors and n-body potential fields. Proceedings 24th Annual ACM Symposium on the Theory of Computing, 1992, pp. 546–556.

[13] P. B. Callahan and S. R. Kosaraju. Algorithms for dynamic closest pair and n-body potential fields. Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 263–272.

[14] Y.-J. Chiang, F. T. Preparata, and R. Tamassia. A unified approach to dynamic point location, ray shooting, and shortest paths in planar maps. Proc. 4th ACM-SIAM Symp. on Discrete Algorithms, 1993, pp. 44-53.

[15] R. F. Cohen and R. Tamassia. Combine and conquer: a general technique for dynamic algorithms. Proc. First Europ. Symp. on Algorithms, Lecture Notes in Computer Science. Springer-Verlag, 1993, pp/ 97–108.

[16] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms, 1993, pp. 175-194.

[17] M. J. Golin, R. Raman, C. Schwarz, and M. Smid. Randomized data structures for the dynamic closest-pair problem. Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms, 1993, pp. 301-310.

[18] M. J. Golin, R. Raman, C. Schwarz, and M. Smid. Simple randomized algorithms for closest pair problems. Proc. 5th Canadian Conf. Comput. Geom., 1993, pp. 246-251.

[19] S. Kapoor and M. Smid. New techniques for exact and approximate dynamic closest-point problems. Proc. 10th Annual ACM Symp. Comput. Geom., 1994, pp. 165–174.

[20] H.-P. Lenhof and M. Smid. Enumerating the k closest pair optimally. Proc. 33rd Ann. IEEE Symp. Found. Comput. Sci., 1992, pp. 380-386.

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

[22] J. S. Salowe. Shallow interdistance selection and interdistance enumeration. International Journal of Computational Geometry & Applications, 2, 1992, pp. 49–59.

[23] M. I. Shamos and D. Hoey. Closest-point problem. Proc. 16th Annual IEEE Symp. Found. Comput. Sci., 1975, pp. 151-162.

[24] C. Schwarz. Data Structures and Algorithms for the Dynamic Closest Pair Problem. Ph.D. Thesis, Universität des Saarbrücken, 1993.

[25] C. Schwarz and M. Smid. An $O(n \log n \log\log n)$ algorithm for the on-line closest pair problem. Proc. 3th ACM-SIAM Symp. Discrete Algorithms, 1992, pp. 280–285.

[26] C. Schwarz, M. Smid, and J. Snoeyink. An optimal algorithm for the on-line closest-pair problem. Proc. 8th Annual ACM Symp. Comput. Geom., 1992, pp. 330–336.

[27] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. Journal of Computer and System Sciences, 26, 1983, pp. 362–391.

[28] M. Smid. Dynamic rectangular point location, with an application to the closest pair problem. Technical Report MPI-I-91-101, Max-Plank-Institut fürInformatik, Saarbrücken, Germany, 1991

[29] M. Smid. Maintaining the minimal distance of a point set in less than linear time. Algorithms Rev., 2, 1991, pp. 33-44.

[30] M. Smid. Maintaining the minimal distance of a point set in polylogarithmic time. Discrete & Computational Geometry, 7, 1992, pp. 415–431.

[31] K. L. Supowit. New techniques for some dynamic closest-point and farthest-point problems. Proc. 1-st Annual ACM-SIAM Symposium on Discrete Algorithms, 1990, pp. 84–90.

[32] P. M. Vaidya. An optimal algorithm for all-nearest-neighbors problem. Proceedings 27th Annual Symposium Found. Comp. Sc., 1986, pp. 117–122.

[33] P. M. Vaidya. An $O(n \log n)$ algorithm for all-nearest-neighbors problem. Discrete Comput. Geom., 1989, pp. 101–115