An optimal algorithm for the on-line closest pair problem

Christian Schwarz   Michiel Smid   Jack Snoeyink

MPI–I–91–123   November 1991

An optimal algorithm for the on-line closest pair problem

Christian Schwarz   Michiel Smid   Jack Snoeyink

MPI–I–91–123   November 1991

"Das diesem Bericht zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministers für Forschung und Technologie (Betreuungskennzeichen ITS 9103) gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor."

In the *off-line* version of the problem, the complete set of points is known at the start of the algorithm. This version of the problem has been solved optimally for a long time. In 1975, Shamos and Hoey [9] gave an $O(n \log n)$ algorithm for the planar case. One year later, Bentley and Shamos [1] gave an $O(n \log n)$ algorithm for the k-dimensional case. See also Vaidya [14], who solved the all-nearest-neighbors problem within the same time bound. All these algorithms can be implemented in the algebraic decision tree model, for which an $\Omega(n \log n)$ lower bound holds. See Preparata and Shamos [6].

In this paper, we consider the *on-line* closest pair problem. Here, the points arrive one after another. After a point arrives, we have to update the current closest pair. This version of the problem has only been studied recently.

In Smid [11], an algorithm is given that computes the closest pair on-line, in $O(n(\log n)^{k-1})$ time. This algorithm only uses algebraic functions. Therefore, it is optimal for the planar case.

Smid [11] and Schwarz and Smid [8] give algorithms that run in $O(n(\log n)^2 / \log\log n)$ and in $O(n \log n \log\log n)$ time, respectively, for any fixed dimension k. These algorithms, however, use the non-algebraic floor function. If additionally the functions EXP and LOG are available at unit-cost, the running time of the algorithm in [8] can be improved to $O(n \log n)$.

In this paper, we give an $O(n \log n)$ algorithm for any fixed dimension k that uses only algebraic functions. Hence, the algorithm is optimal. More precisely, we give a data structure that maintains the closest pair in $O(\log n)$ amortized time per insertion. Our structure can also solve the problem of computing on-line the closest pair that existed over the history of a fully dynamic point set in $O(\log n)$ amortized time per insertion or deletion.

Note that recently there has been much interest in the dynamic closest pair problem. For the case, where only deletions are allowed, see Supowit [13]. For the fully dynamic case, see Smid [10, 12], Salowe [7] and Dickerson and Drysdale [3].

The algorithm in this paper is based on the algorithm of Smid [11]. To update the closest pair when a point is inserted, that algorithm makes some queries into a data structure for the k-dimensional rectangular point location problem. In this data structure, one query takes $O((\log n)^{k-1})$ time, which causes the entire algorithm to have an amortized insertion time of $O((\log n)^{k-1})$.

In this paper, we also use a data structure for the rectangular point location problem. The subdivisions of k-space that arise, however, are regular enough to allow point location queries to be solved in logarithmic time. The data structure for these queries is implemented using centroids and tree decompositions. Chazelle [2] introduced such decompositions to computational geometry with his polygon cutting theorem. Guibas et al. [4] gave a procedure to compute them in linear time.

In Section 2, we give the basic algorithm for maintaining the closest pair under insertions. We define the subdivision that is used during this algorithm, and give an initial data structure that implements the insertion operation using point location.

In Section 3, we use centroids for the implementation of point location. In this way, the time for one query is improved to $O(\log n)$. (In Section 2, this time could be linear.) In order to maintain this improved data structure, we use the partial rebuilding technique. (See e.g. Overmars [5].)

In Section 4, we apply our solution to computing the closest pair in history and give some concluding remarks.

2 The basic algorithm

In this section, we give a data structure that maintains the closest pair in a point set under insertions of points. The basic idea is the same as in Smid [11]. We give all details, however, to keep the paper self-contained.

The algorithm maintains a subdivision of k-space into axes-parallel hyperrectangles, called k-boxes for short. Formally, a *k-box* has the form

$$[a_1 : b_1] \times [a_2 : b_2] \times \ldots \times [a_k : b_k],$$

where $a_i \in \mathbb{R} \cup \{-\infty\}$, $b_i \in \mathbb{R} \cup \{\infty\}$ and $a_i < b_i$ for $i = 1, \ldots, k$.

We say that a point $p = (p_1, p_2, \ldots, p_k)$ is *contained* in the above k-box, if $a_i \leq p_i < b_i$ for all i. In this way, even if a point lies on the boundary of many k-boxes, the notion of containment is uniquely defined.

The data structure: The essential component of the closest-pair data structure is a hierarchical subdivision of space into k-boxes. Let V be the current set of points, and let n be its cardinality. The data structure stores the following information:

- A pair of points (P, Q) that are a closest pair in V and a variable δ whose value is the distance $d(P, Q)$.
- A binary tree T representing the current subdivision of k-space. The nodes of T store k-boxes, where the k-boxes stored in the leaves form a subdivision of k-space.

  For each non-leaf node v, the k-box stored in it is equal to the union of the two k-boxes that are stored in the two children of v.
- With each leaf of T, we store a list of all points in V that are contained in the k-box stored in this leaf. (These points are stored in an arbitrary order.)

The k-boxes stored at the leaves of this data structure have some additional constraints that we enforce.

(1) each leaf k-box has sides of length at least δ, where δ is the distance of the closest pair in V.

(2) each k-box contains at least one and at most $(2k + 2)^k$ points of V.

(3) all k-boxes are non-overlapping and together they partition the entire k-space.

Initializing the structure: Suppose that we start with a set V of size two. Then the initial subdivision of k-space consists of one k-box, namely the entire space. The binary tree T consists of one leaf node, whose k-box is the only box in the subdivision.

With this leaf, we store a list containing the two points. The pair (P, Q) stores the two points, and the value of δ is equal to its distance.

Clearly, after the initialization, the subdivision and the data structure satisfy the above constraints.

Our algorithm to insert a new point will use *point location* as a subroutine. Thus, before giving the algorithm we describe a simple-minded method to use the binary tree T to answer point location queries in linear time. In Section 3 we improve point location to logarithmic time.

Point location: Let p be a point in k-space. In a point location query, we have to locate the k-box in the current subdivision that contains p. This query is answered as follows.

Starting in the root of the tree T, we visit the nodes of T on the path to the leaf whose k-box contains p. We maintain as an invariant that p is contained in the k-box that is stored in the current node. Suppose we have reached the non-leaf node v. Point p is contained in exactly one of the k-boxes that are stored in the two children of v. The search proceeds in the child storing this k-box.

The procedure ends if we reach a leaf. By the invariant, the k-box stored in this leaf contains the query point p.

The insertion algorithm: Let $p = (p_1, \ldots, p_k)$ be the point to be inserted. The algorithm makes two steps. The first step updates the closest pair; the second updates the rest of the data structure.

1. Update the closest pair: Observe that only boxes intersecting the δ-ball around the new point p can contain points q such that $d(p,q) < \delta$. Therefore, we first identify these boxes. For this purpose, we perform 3^k point location queries, with query points

$$(p_1 + \epsilon_1, \ldots, p_k + \epsilon_k), \text{ for } \epsilon_1, \ldots, \epsilon_k \in \{-\delta, 0, \delta\}.$$

Then, for each k-box that is located, we walk through its list of points. For each point q that is in one of these lists, if $d(p,q) < \delta$, we set $(P, Q) := (p, q)$ and $\delta := d(p,q)$.

2. Update the rest of the data structure: In the previous step, we have located the leaf v of the binary tree T whose k-box contains point p. We insert p into the list that is stored with v.

If afterwards this list contains at most $(2k+2)^k$ points, the algorithm is finished. That is, the subdivision is not changed.

Otherwise, if it contains $1 + (2k+2)^k$ points, we perform a split operation on the k-box stored in v. This split operation is defined as follows.

Suppose we want to split the k-box $B = [a_1 : b_1] \times \ldots \times [a_k : b_k]$ of the current subdivision. Let V' be the set of points that are stored in the list of B.

For $i = 1, \ldots, k$, we compute the values m_i and M_i, which are, respectively, the minimal and maximal i-th coordinate of any point of V'. Let i be an index such that $M_i - m_i > 2\delta$. (In Lemma 2, we show that there is such an index.) Let $c_i := m_i + (M_i - m_i)/2$.

Then we split the k-box B into two k-boxes

$$B_l = [a_1 : b_1] \times \ldots \times [a_{i-1} : b_{i-1}] \times [a_i : c_i] \times [a_{i+1} : b_{i+1}] \times \ldots \times [a_k : b_k]$$

and

$$B_r = [a_1 : b_1] \times \ldots \times [a_{i-1} : b_{i-1}] \times [c_i : b_i] \times [a_{i+1} : b_{i+1}] \times \ldots \times [a_k : b_k].$$

In the tree T, the leaf v corresponding to B gets two children, one child for the k-box B_l and one for the k-box B_r. The list that is stored with v is removed, and it is split in two lists for the new leaves.

This concludes the insertion algorithm. First, we prove a sparseness result that is needed in the proof of Lemma 2.

Lemma 1 *Let V be a set of points in k-dimensional space, and let δ denote the minimal distance in V. Then any k-dimensional cube having sides of length 2δ contains at most $(2k+2)^k$ points of V.*

Proof: Partition the cube into $(2k+2)^k$ subcubes with sides of length $\delta/(k+1)$. Now assume that the cube contains at least $(2k+2)^k + 1$ points of V. Then one of the subcubes contains at least two points of V. These two points have a distance that is at most equal to the L_t-diameter of this subcube. This diameter, however, is at most $k \cdot \delta/(k+1) < \delta$. This contradicts the fact that the minimal distance of V is δ. ∎

In the next lemma, we show that the index i that is used in the split operation indeed exists.

Lemma 2 *Let V be a set of points in k-space, and let δ be the distance of a closest pair in V. Let B be a k-box that contains more than $(2k+2)^k$ points of V. For $i = 1, \ldots, k$, define the minimal, m_i, and maximal, M_i, i-th coordinates of any point in $V \cap B$. Then there is an index i, such that $M_i - m_i > 2\delta$.*

Proof: Assume that $M_i - m_i \leq 2\delta$ for all $i = 1, \ldots, k$. Then, there is a k-cube B' having side lengths 2δ that contains all points of $V \cap B$. By the previous lemma, however, the cube B' contains at most $(2k+2)^k$ points of V. This is a contradiction. ∎

Lemma 3 *Let B be a k-box in the subdivision of k-space whose list contains $1+(2k+2)^k$ points of V. Let δ be the minimal distance of V. Suppose, we perform a split operation on B. After this operation, the sides of the two newly created k-boxes have length at least δ, and each such k-box contains at least one and at most $(2k+2)^k$ points of V.*

Proof: The lemma follows immediately from the split algorithm. ∎

Lemma 4 *The insertion algorithm correctly maintains the closest pair data structure.*

Proof: Let δ be the minimal distance just before the insertion of point p. If this minimal distance changes, there must be a point inside the L_t-ball of radius δ centered at p. This ball is contained in the k-box $[p_1 - \delta : p_1 + \delta] \times \ldots \times [p_k - \delta : p_k + \delta]$. Therefore, it suffices to compare p with all points of the current set V that are in this box. Let

$$W := V \cap ([p_1 - \delta : p_1 + \delta] \times \ldots \times [p_k - \delta : p_k + \delta])$$

be the set of these points, and let W' be the set of points that are contained in the lists corresponding to the k-boxes that result from the 3^k point location queries. The algorithm compares p with all points in W'. Hence, if we show that $W \subseteq W'$, then it is clear that the algorithm correctly maintains the closest pair.

Let $q = (q_1, \ldots, q_k)$ be a point in W. Assume w.l.o.g. that $q_i \geq p_i$ for $i = 1, \ldots, k$. Then $p_i \leq q_i \leq p_i + \delta$ for $i = 1, \ldots, k$. Let B be the k-box in the current subdivision of k-space whose list contains q. Assume that $q \notin W'$. Then B does not contain any of the 2^k points $(p_1 + \alpha_1, \ldots, p_k + \alpha_k)$, where $\alpha_1, \ldots, \alpha_k \in \{0, \delta\}$. These 2^k points are the corners of the k-box

$$B' := [p_1 : p_1 + \delta] \times \ldots \times [p_k : p_k + \delta],$$

having sides of length δ. (Note that in general B' is not part of the current subdivision of k-space.) Since $q \in B'$, and since B does not contain any of the corner points of B', it follows that the box B must have at least one side of length strictly less than δ. This contradicts the definition of our data structure. Hence, $q \in W'$ and, therefore, $W \subseteq W'$. This proves that the insertion algorithm correctly maintains the closest pair.

It remains to show that the new subdivision satisfies the invariants (1)–(3). Consider a k-box of the current subdivision that is not split during the insertion. Since the value of δ can only decrease, the side lengths of this box remain at least equal to δ. Clearly, if the box contains at least one point before the insertion, so it does afterwards. Also, the box still contains at most $(2k+2)^k$ points.

If a k-box is split, then Lemma 3 guarantees that the new k-boxes have sides of length at least δ, that they contain at least one and at most $(2k+2)^k$ points. Finally, it is clear that the k-boxes that are not split, together with the two new k-boxes, are non-overlapping and partition k-space. ∎

The central operations of the insertion algorithm are point location and splitting a k-box of the subdivision. The following theorem expresses the running time of the algorithm in terms of the cost of these two operations.

Theorem 1 *Let $Q(n)$ be the time for one point location query and $S(n)$ be the time for one split operation. The given data structure has linear size and maintains the closest pair of the set V in $O(Q(n) + S(n))$ time per insertion.*

Proof: The binary tree T has at most n leaves, because each leaf corresponds to a non-empty k-box. Therefore, T has linear size. Since any point is stored in exactly one list, all these lists together also have linear size. This proves the space bound.

Consider the insertion algorithm. We need $O(3^k Q(n))$ time for all point location queries. Then, we walk through at most 3^k lists, each of which has size at most $(2k+2)^k$. This takes time $O(3^k(2k+2)^k)$.

In case no split operation is necessary, the data structure needs $O(1)$ time to update the rest of the data structure. Otherwise, we need $S(n)$ time for the split operation.

It follows that the overall insertion time is bounded by

$$O(3^k Q(n) + 3^k(2k+2)^k + S(n)),$$

which is $O(Q(n) + S(n))$, because k is a constant. ∎

Let h denote the height of the binary tree T. Then, clearly, it takes $O(h)$ time to solve one point location query. Since h can be linear in n, it follows that $Q(n)$ can be $\Theta(n)$. Consider a split operation. First, it takes $O(k(2k+2)^k)$ time to find the index i. Then, the operation can be completed within the same time bound. Hence, since k is a constant, $S(n) = O(1)$. Therefore, an insertion takes $O(n)$ time in the worst-case.

In the next section, we build an additional search structure on T that improves the point location time, $Q(n)$, to $O(\log n)$. In order to maintain the search structure, we increase the split time, $S(n)$, to $O(\log n)$ in the amortized sense. Hence, it follows from Theorem 1 that the insertion algorithm will need $O(\log n)$ amortized time for one insertion.

3 Point location using the tree decomposition

In the previous section, the point location algorithm started in the root of the tree T and followed a path until it reached a leaf. We observe, however, that it is not necessary to start in the root; the algorithm can start in an arbitrary node.

Suppose we start in node v. Let B_v be the k-box that is stored with v. If the query point p is contained in B_v, the search continues in one of the two subtrees rooted at children of v. Otherwise, if p is not contained in B_v, the search continues in the tree that is obtained by removing the subtree rooted at v. These searches proceed recursively, i.e., again they do not necessarily start in the root of the subtree. If we choose our initial node v such that the two subtrees have roughly equal size and repeat choosing nodes in this way for the recursive searches, then we get a logarithmic search time.

In the rest of this section we do three things. First, we define the *β-decomposition tree* T_β on the nodes of the tree T. A procedure of Guibas et al. [4] can be used to compute β-decomposition trees suitable for our purpose. Second, we define our new data structure that uses the tree decomposition. Third, with this new data structure, we implement the two central operations of our on-line closest pair algorithm (cf. the remark preceding Theorem 1): we show how to do logarithmic-time point location in the subdivision of k-boxes, and we show how to do a split operation on a leaf of the tree representing the subdivision.

We call an internal node $v \in T$ a β-centroid if the removal of v results in three connected components, each containing at most $\beta|T|$ nodes. (Here, $|T|$ denotes the number of nodes in T.) Notice that a β-centroid is also a β'-centroid for all $\beta' \geq \beta$. A β-decomposition tree of T, denoted T_β, is defined recursively: The β-decomposition tree of a leaf is just the leaf. Otherwise, the root of T_β is a β-centroid $v \in T$, and the children are β-decomposition trees for the connected components of $T - v$. The trees T and T_β have the same set of leaves and the same set of internal nodes.

Since T is binary, the β-decomposition tree T_β is ternary. For any node $v \in T_\beta$ we have three pointers, *left*(v), *right*(v), and *up*(v), that point to the β-decomposition trees for the connected components of $T - v$ that contain the left child of v in T, the right child of v in T, and the parent of v in T, respectively. The nodes that are stored in the subtree of T_β rooted at v form the *component of* v, denoted by $C(v)$. From the decomposition scheme, we have $C(v) = C(\mathit{left}(v)) \mathbin{\dot\cup} C(\mathit{right}(v)) \mathbin{\dot\cup} C(\mathit{up}(v)) \mathbin{\dot\cup} \{v\}$. Finally, we note that the depth of T_β is $O(\log_{(1/\beta)} |T|)$.

In [4], Guibas et al. give an algorithm that computes a centroid decomposition of a binary tree T in linear time. In that paper, the tree T is decomposed by removing a *centroid edge* which decomposes T into two parts, each of size at least $\lfloor (|T| + 1)/3 \rfloor$. A straightforward modification of their algorithm, however, also computes a β-decomposition tree T_β for T in linear time, with $\beta = 1/2$.

The improved data structure: As in Section 2, the data structure comprises the closest pair and a tree T storing a subdivision of k-space into k-boxes, whose leaves store the current subdivision and satisfy (1)–(3). Each internal node v stores the union of the k-boxes stored in the leaves of the subtree rooted at v. For each leaf x, there is a list of the points of V lying in the k-box stored at x.

We also maintain a ternary *β-decomposition tree* T_β of T, where $\beta = 3/4$. As described above, $C(v)$, the component of v, consists of the nodes in the subtree of T_β rooted at v. With each node $v \in T_\beta$, we store the size of $C(v)$.

Point location: The β-decomposition tree T_β guides point location in the tree T that represents the subdivision of k-space into k-boxes.

Let p be a point in k-space, and let s be the unique leaf in T whose k-box contains p. Our task is to find s. The algorithm consecutively checks nodes v, starting in the root of the decomposition tree T_β. We maintain the invariant that, if v is the current node, then $s \in C(v)$. At the start of the algorithm, when v is the root of T_β, the invariant is trivially true, since in this case all nodes are in $C(v)$.

Now let v be the current node. By induction, we assume that the invariant holds for v, which means that $s \in C(v)$. If v is a leaf of T, then the invariant implies that $v = s$, and we are done. If v is not a leaf of T, then $v \neq s$. Since $s \in C(v)$ by the invariant, we have $|C(v)| > 1$ in this case. This means that the subtree of v in T_β has more than one node, which in turn implies that v is not a leaf in T_β. Let $x = \mathit{left}(v)$, $y = \mathit{right}(v)$, and $z = \mathit{up}(v)$ be the children of v in T_β. We know that at least one of the nodes x, y, z exists.

We examine v as follows: in constant time, we check whether point p is inside B_v, the k-box corresponding to v. If p is inside B_v, then we check which one of the two k-boxes of children of v in T also contains p. With this knowledge, we can choose the correct child of v in T_β to continue the search: If p lies in the box stored in the left child of v in T, then s must be in the left subtree in T. In this case, we choose x to be the new current node. Since the only part of $C(v)$ which lies in the left subtree of v in T is $C(x)$, we have $s \in C(x)$. The case that p lies in the right subtree of v in T is symmetric. It remains to consider the case $p \notin B_v$. Here, we choose z to be the new current node. Since $p \notin B_v$, we know that $s \notin C(x)$ and $s \notin C(y)$, which implies $s \in C(z)$.

The search proceeds via edges of T_β and ends in a leaf of T_β. From our invariant, this leaf must be s. Therefore we have the following lemma:

Lemma 5 *Let T be a binary tree of size n, storing a collection of k-boxes in k-space as defined in the previous section, and let p be a query point. Given a β-decomposition tree of T, point location, i.e., identifying the k-box containing p, can be done in $O(\log_{(1/\beta)} n)$ time.*

From the definition of the improved data structure, we have $\beta = 3/4$, and it follows that $Q(n) = O(\log n)$. Next, we discuss how to maintain the tree T_β if split operations are performed. We shall show that the improved data structure can be correctly maintained, and that $S(n) = O(\log n)$ amortized.

Split operation: Let x be a leaf of T and suppose that we perform a split operation on the k-box stored in x. Then, x is turned into an internal node and is given two new children x_1, x_2, in T as well as in T_β.

Updates gradually unbalance the β-decomposition tree T_β; we must maintain T_β in amortized logarithmic time. For each node $v \in T_\beta$, we store the size of $C(v)$, as prescribed in the definition of the improved data structure. When we add leaves to T, the nodes of T_β whose counts change are those on the search path to the leaves—we can update these counts in $O(\log n)$ time. Then we can determine the highest node in T_β that is no longer a $(3/4)$-centroid and rebuild its subtree in T_β. Using the algorithm of [4], we compute a $(1/2)$-decomposition for this subtree in time proportional to its size.

If we build the subtree of T_β rooted at v, then v is a $(1/2)$-centroid. In order to rebuild v's subtree, one must increase the size of the subtree by a quarter. Thus, if every leaf inserted into the subtree brings along a credit, we can use these credits to pay for the rebuilding. Since the depth of T_β is $O(\log n)$, each additional leaf needs only $O(\log n)$ credits. It follows that the total cost of rebuilding over n insertions is $O(n \log n)$. We have the following lemma:

Lemma 6 *The amortized cost of a split operation is $O(\log n)$.*

From Lemma 5 and Lemma 6, we have point location cost $Q(n) = O(\log n)$ and split cost $S(n) = O(\log n)$ amortized. Combining this with Theorem 1 gives the following result.

Theorem 2 *The improved data structure has linear size and maintains the closest pair in the point set V in $O(\log n)$ amortized time.*

Corollary 1 *The closest pair in a set of n points in k-dimensional space can be computed on-line in $O(n \log n)$ time, using $O(n)$ space. This is optimal in the algebraic decision tree model.*

4 The "closest pair in history" and open problems

We have given an optimal solution to the problem of maintaining a closest pair as points are inserted on-line. It is natural to ask about fully dynamic point sets in which points can be inserted and deleted.

While we cannot efficiently maintain the closest pair under on-line insertions and deletions, we can solve the problem of recording the closest pair in history—recording the closest pair of points that existed simultaneously during an on-line sequence of point insertions and deletions. Such a record could help verify that an appropriate step size was used in a dynamic system simulation.

Theorem 3 *The closest pair of points over the history of a sequence of insertions and deletions can be computed on-line in $O(n \log n)$ time, using $O(n)$ space.*

Proof: We use the improved data structure with the insertion operation specified above. To delete a point p, we locate the leaf node v whose k-box B_v contains p. We delete p from v's point list in constant time. If some points remain in B_v then we are done—the invariants still hold. (Note that δ can only decrease, so there is no problem with the side lengths of the k-boxes.)

Otherwise, we must delete the node v and contract the parent and sibling of v into one node. As in the splitting algorithm, we update the component counts that change in T_β in $O(\log n)$ time. Then we can rebuild the subtree of the highest node in T_β that is no longer a $(3/4)$-centroid.

Since rebuilding forms a $(1/2)$-decomposition, the number of updates between two rebuildings at a node is still proportional to the size of its subtree. Thus, if each deletion also comes with $O(\log n)$ credits to give to the nodes on the search path in T_β, these credits can be used to pay for the rebalancing. Thus, deletion costs also amortize to $O(\log n)$ per deletion. ∎

Other open problems remain. In some applications, one would like to change the amortized time bounds for insertions to worst-case bounds. Algorithms using floors, randomness, or other models of computation may have $o(n \log n)$ running times.

References

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

[2] B. Chazelle. *A theorem on polygon cutting with applications.* Proc. 23rd Annual IEEE Symp. on Foundations of Computer Science, 1982, pp. 339-349.

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

[4] L. Guibas, J. Hershberger, D. Leven, M. Sharir and R.E. Tarjan. *Linear time algorithms for visibility and shortest path problems inside triangulated simple polygons.* Algorithmica **2** (1987), pp. 209-233.

[5] M.H. Overmars. *The Design of Dynamic Data Structures.* Lecture Notes in Computer Science, Vol. 156, Springer-Verlag, Berlin, 1983.

[6] F.P. Preparata and M.I. Shamos. *Computational Geometry, an Introduction.* Springer-Verlag, New York, 1985.

[7] J.S. Salowe. *Shallow interdistance selection and interdistance enumeration.* Proc. WADS'91, LNCS Vol. 519, Springer-Verlag, Berlin, 1991, pp. 117-128.

[8] C. Schwarz and M. Smid. *An $O(n \log n \log\log n)$ algorithm for the on-line closest pair problem.* To appear in: Proceedings 3rd Annual ACM-SIAM Symp. on Discrete Algorithms, 1992.

[9] M.I. Shamos and D. Hoey. *Closest-pair problems.* Proc. 16th Annual IEEE Symp. on Foundations of Computer Science, 1975, pp. 151-162.

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

[11] M. Smid. *Dynamic rectangular point location, with an application to the closest pair problem.* Report MPI-I-91-101, Max-Planck-Institut für Informatik, Saarbrücken, 1991. See also: Proc. 2nd Annual International Symp. on Algorithms, 1991.

[12] M. Smid. *Maintaining the minimal distance of a point set in polylogarithmic time (revised version).* Report MPI-I-91-103, Max-Planck-Institut für Informatik, Saarbrücken, 1991. See also: Proc. 2nd Annual ACM-SIAM Symp. on Discrete Algorithms, 1991, pp. 1-6.

[13] K.J. Supowit. *New techniques for some dynamic closest-point and farthest-point problems.* Proc. 1st Annual ACM-SIAM Symp. on Discrete Algorithms, 1990, pp. 84-90.

[14] P.M. Vaidya. *An $O(n \log n)$ algorithm for the all-nearest-neighbors problem.* Discrete Comput. Geom. **4** (1989), pp. 101-115.