The colored quadrant priority search tree with an application to the all-nearest-foreign-neighbors problem

Andreas Brinkmann, Thorsten Graf, Klaus Hinrichs

Institut für numerische und instrumentelle Mathematik -INFORMATIK-
Westfälische Wilhelms-Universität Münster
Einsteinstr. 62 D-48149 Münster e-mail: graf@math.uni-muenster.de khh@math.uni-muenster.de

EXTENDED ABSTRACT

Abstract. Consider a dynamic set of points in the plane having different colors. Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be two keys according to which points may be sorted, e.g. the x- and the y-coordinate. A point $p \in S$ is called *proper* with respect to a query point $p_0 \in S$ iff $\mathcal{K}_1(p_0) < \mathcal{K}_1(p)$, $\mathcal{K}_2(p_0) < \mathcal{K}_2(p)$, and if p has a different color than p_0.
We present the *colored quadrant priority search tree* (CQPST) which supports the efficient search for all k proper points in $O(\log n + k)$ time and for the minimal proper point with respect to key $\mathcal{K}_2$ in $O(\log n)$ time.
As an application we show that the CQPST can be used to solve the all-nearest-foreign neigbors problem with respect to any arbitrary L^t-metric ($1 \leq t \leq \infty$) in optimal $O(n \log n)$ time.

1 The colored quadrant priority search tree

Let S be a dynamic set of points in the plane having different colors, and let $c(p)$ denote the color of a point $p \in S$. Let $\mathcal{K}_1$ and $\mathcal{K}_2$ be two keys according to which points in the plane may be sorted, e.g. the x-coordinate, the y-coordinate, or the sum of the coordinate values. For a point p in the plane $\mathcal{K}_1(p)$ and $\mathcal{K}_2(p)$ denote the values of p with respect to $\mathcal{K}_1$ and $\mathcal{K}_2$, respectively. For ease of presentation we assume that at any time the set S does not contain any two points which have equal $\mathcal{K}_1$- or $\mathcal{K}_2$-values. This restriction is not essential and can be easily overcome.

Let p and p_0 be two points in S. Then p is called *proper* with respect to p_0 iff $\mathcal{K}_1(p_0) < \mathcal{K}_1(p)$, $\mathcal{K}_2(p_0) < \mathcal{K}_2(p)$, and p and p_0 have different colors. The set of proper points for a point p_0 is denoted by $PP(p_0)$. We consider the following two problems: For a query point $p_0 \in S$ determine 1) the set $PP(p_0)$ and 2) the minimal point in $PP(p_0)$ with respect to $\mathcal{K}_2$. If $PP(p_0)$ is empty the queries should return *nil*. The examples in Figure 1 show for different keys $\mathcal{K}_1$ and $\mathcal{K}_2$ the ranges which contain the proper points with respect to p_0. Different colors are indicated by different shadings of the points. A circled point is the minimal proper point in $PP(p_0)$ with respect to $\mathcal{K}_2$.

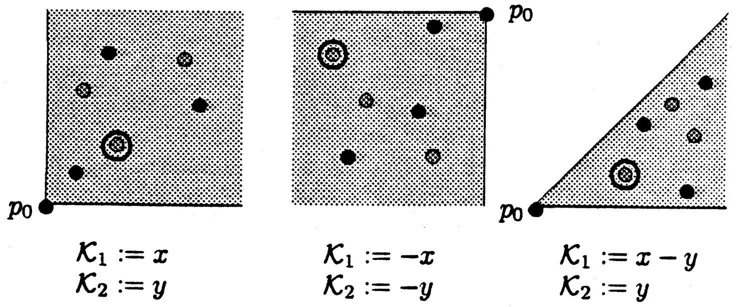

Figure 1: Ranges for keys $\mathcal{K}_1$ and $\mathcal{K}_2$

In this paper we present the *colored quadrant priority search tree* (CQPST) which supports the following operations:

1. *insert(p)*: Insert the point p into the CQPST.
2. *delete(p)*: Remove the point p from the CQPST.
3. *AllProper(p_0)*: Return the set $PP(p_0)$ of proper points or the *nil*-set.
4. *MinProper(p_0)*: Report the point in $PP(p_0)$ which is minimal with respect to $\mathcal{K}_2$ or the *nil*-point.

1.1 Structure of the CQPST

The CQPST is based on the *priority search tree* which supports an efficient three sided range query ([9], [6]). The operation *AllProper* of the CQPST is similar to the three sided range query supported by the priority search tree. However, since priority search trees cannot distinguish points with respect to their color they do not support the operations *MinProper* and *AllProper*.

The skeleton of the CQPST is a ***binary tree with constant linkage cost per update*** (CLC-tree) ([11]) which requires $O(\log n)$ update time with only $O(1)$ rotations. This rotation property is of great importance since each rotation itself may cause an $O(\log n)$ time operation. A unifying framework for CLC-trees can be found in [11]. Special CLC-tree schemes are e.g. the ***half balanced tree*** ([10]) and the ***red-black tree*** ([4, 15, 2]).

The CQPST is a 0-2 binary tree, i.e. each inner node has exactly two sons, and a leaf search tree for the $\mathcal{K}_2$-values in S, i.e. for every value $\mathcal{K}_2(p)$ $(p \in S)$ there exists one leaf in the tree. Each node contains space to store a point which is possibly the *nil*-point. In each node k of the CQPST we additionally store the following information:

I1. The split value $sv(k)$ which is the maximum $\mathcal{K}_2$-value stored in a leaf of its left subtree $LST(k)$.

I2. The maximal $\mathcal{K}_1$-value $Max(k)$ of the points stored in the subtree rooted by the node k which have a color different from the color of the point stored in k.

The value of $Max(k)$ is essential for finding *MinProper*(p_0) for a query point $p_0 \in S$ stored in node k. The points are stored in the nodes of the CQPST according to the following three conditions:

C1. Each point p lies on the path from the root to the leaf with value $\mathcal{K}_2(p)$.

C2. The $\mathcal{K}_1$-values of the points stored along an arbitrary root-to-leaf path are in decreasing order.

C3. If a node contains a point then its father does, too.

Condition C2 implies a ***heap-property*** of the CQPST with respect to the $\mathcal{K}_1$-values, i.e. for each subtree of the CQPST the root node is the node with the largest $\mathcal{K}_1$-value. Note that $\mathcal{K}_2(p)$ (except for the maximal $\mathcal{K}_2(p)$) is the split-value of the node encountered after the first right turn on the path from the leaf containing $\mathcal{K}_2(p)$ to the root. Condition C1 implies that at most one of two sibling leaves may contain a point. In the next two sections we show that the values $sv()$ and $Max()$ stored in the nodes and the three conditions (C1, C2, and C3) can be maintained during insertions, deletions and rebalancing operations as *invariants* of the CQPST.

1.2 Inserting and deleting

A point p is *inserted* into the CQPST by first inserting $\mathcal{K}_2(p)$ and performing the necessary rebalancing operations. As we will show in section 1.3 the CQPST can be rebalanced in optimal $O(\log n)$ time. Then we sift the point down the tree according to its weight, i.e. $\mathcal{K}_1(p)$: Compare the $\mathcal{K}_1$-values of p and the point stored in the root node, store the point with the larger $\mathcal{K}_1$-value and continue this operation with the other point, exploring the root of the left or right subtree depending on the $\mathcal{K}_2$-value of this point, until an empty node is reached. It is not difficult to prove that there exists an empty node k^* which terminates this process. All invariants except I2 can be easily maintained ([6]). I2 is maintained as follows: $Max()$ may only change for nodes in the path from the root to the node k^* and is updated during the sift-down process: Let p be a point to be sifted down and q be the point stored in the node k under consideration:
If $c(p) = c(q)$ then $Max(k)$ does not change. If $c(p) \neq c(q)$ then we have to distinguish two cases:

a) $\mathcal{K}_1(p) < \mathcal{K}_1(q)$:
The point p is sifted down and q remains in node k. If $\mathcal{K}_1(p) > Max(k)$ then we update $Max(k) := \mathcal{K}_1(p)$.

b) $\mathcal{K}_1(p) > \mathcal{K}_1(q)$:
The point q is sifted down and p is stored in node k. Due to the heap property of the CQPST the point q is the $\mathcal{K}_1$-maximal point stored below p with different color than p, and we update $Max(k) := \mathcal{K}_1(q)$.

This shows that after the insertion process all invariants of the CQPST are maintained correctly.

To *delete* a point p first search for it and remove it from its node. Fill the gap by sifting successor points up the tree without changing the tree structure. The $\mathcal{K}_1$-heap property is maintained by pushing up the point contained in the son nodes which has the larger $\mathcal{K}_1$-value. This process is continued until we arrive at a node k' which has two empty son nodes or is a leaf. Finally delete the leaf containing $\mathcal{K}_2(p)$. Again, all invariants except of I2 can be easily maintained. The invariant I2 can be maintained as follows: The only nodes for which $Max()$ may change are the nodes along the path K from the root to k'. The $Max()$-values in the nodes of K are updated bottom-up. Let k be a node of K storing a point q and assume that the $Max()$-values of the nodes below k have already been updated. Denote by q_l and q_r the points stored in the left son node k_l and the right son node k_r of k, respectively. We distinguish three cases:

1. $q_l = nil$ and $q_r = nil$
   $Max(k)$ is undefined since there are no points stored in the subtree rooted by k.

2. $q_l \neq nil$ xor $q_r \neq nil$
   Let q^* be the point stored in the non-empty node $k^* \in \{k_l, k_r\}$. If $c(q) = c(q^*)$ then we update $Max(k) := Max(k^*)$, otherwise $Max(k) := \mathcal{K}_1(q^*)$.

3. $q_l \neq nil$ and $q_r \neq nil$

$$\text{If} \left.\begin{array}{l} c(q_l) = c(q) = c(q_r) \\ c(q_l) \neq c(q) = c(q_r) \\ c(q_l) = c(q) \neq c(q_r) \\ c(q_l) \neq c(q) \neq c(q_r) \end{array}\right\} \text{we update } Max(k) := \left\{\begin{array}{l} \max\{Max(k_l), Max(k_r)\} \\ \max\{Max(k_r), \mathcal{K}_1(q_l)\} \\ \max\{Max(k_l), \mathcal{K}_1(q_r)\} \\ \min\{\mathcal{K}_1(q_l), \mathcal{K}_1(q_r)\} \end{array}\right.$$

An inductive argument shows that after a bottom-up walk of the path K all nodes in the CQPST store correct $Max()$-values and all invariants hold. To complete the deletion operation we finally rebalance the tree if necessary.

1.3 Rebalancing the CQPST

It can be easily shown that the invariants of the CQPST can be maintained when rebalancing the tree with *local rebalancing operations*. From the description of the ordinary priority search tree in [6] it follows that the conditions C1, C2, and C3 and invariant I1 can be maintained during local rebalancing operations. In the previous section we showed how to update the $Max()$-values bottom up after the deletion of a point. It is easy to see that the same process applies to restore the $Max()$-values. If all $\mathcal{K}_2$-values are known in advance rebalancing can be avoided by basing the CQPST on the skeleton priority search tree ([12]).

1.4 Performing the queries

The following Lemmas give some properties of the CQPST which we will use to realize the operations *AllProper* and *MinProper*. Let $LST(k)$ denote the left and $RST(k)$ the right subtree of k. The following lemmas are easy to prove, therefore the proofs have been omitted:

Lemma 1.1. *Let k be an inner node of the CQPST, then the following holds:*

1. *For all points $q \in LST(k)$: $\mathcal{K}_2(q) \leq sv(k)$.*

2. *For all points $q \in RST(k)$: $\mathcal{K}_2(q) > sv(k)$.*

Let p_0 be a query point. Let $K := \{r = k_1, \ldots, k_m\}$ be the path from the root r of the CQPST to the first node k_m with split value $\mathcal{K}_2(p_0)$. If p_0 is the point with the largest $\mathcal{K}_2$-value then this is the right-most leaf of the CQPST. Otherwise k_m is an inner node. A node $k_\nu \in K$ $(1 \leq \nu \leq m-1)$ is called a *branch node* iff k_{n+1} is the left son of k_ν. The node k_m is a branch node by definition.

Lemma 1.2. *Let the path K be as described above. Then all proper points are either stored along the path K or in the right subtrees rooted by branch nodes.*

Lemma 1.3. *The proper point with minimal $\mathcal{K}_2$-value is either stored along the path K or in $RST(k_\nu)$ of the deepest branch node k_ν having a right subtree with at least one proper point.*

Figure 2 shows an example with branch nodes k_1 and k_4. The left subtrees which need not be considered are shaded. In the example the point $MinProper(p)$ of the point p stored in r is found in the hilited node.

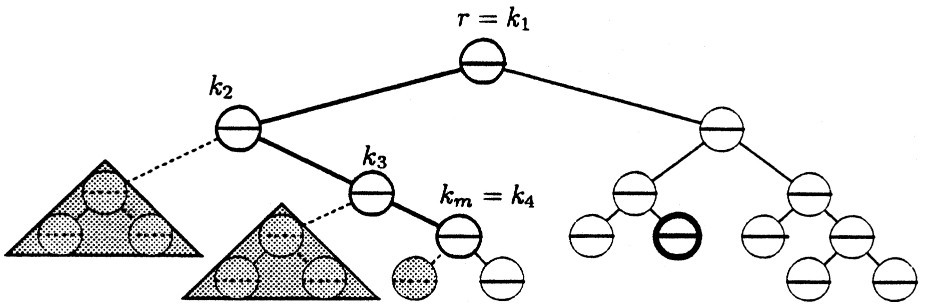

Figure 2: Path $K = \{r = k_1, k_2, k_3, k_4 = k_m\}$

To perform the operation *MinProper* we process the path K in reverse order as follows. Starting in node k_m determine for each branch node k_ν the smallest proper point in $RST(k_\nu)$ or determine in time $O(1)$ the non-existence of such a point:
Start in the right son node k of k_ν. If the point q stored in k satisfies $\mathcal{K}_1(q) \leq \mathcal{K}_1(p_0)$ then the heap property C2 of the CQPST implies that this is also true for all other points in $RST(k_\nu)$. There exists no proper point in $RST(k_\nu)$ and we continue with $k_{\nu-1}$.
If the point q stored in node k satisfies $\mathcal{K}_1(q) > \mathcal{K}_1(p_0)$ and $c(q) \neq c(p)$ then a proper point has been found. If $\mathcal{K}_1(q) > \mathcal{K}_1(p_0)$ and $c(p) = c(q)$ then a proper point is contained in $RST(k_\nu)$ iff $Max(k) \geq \mathcal{K}_1(p_0)$. Hence we can decide in $O(1)$ time whether or not a subtree of the CQPST contains a proper point.
If a proper point is contained in $RST(k_\nu)$ then we tentatively continue the process in the left subtree of k. If a proper point is contained in this subtree then this point is better than any proper point possibly contained in the right subtree of k. If it turns out that this left subtree does not contain a proper point, which can be detected in $O(1)$ time, we continue with the right subtree of k for which we know that it stores at least one proper point. The process stops if we cannot choose a subtree containing a proper point any more.

By Lemma 1.3 we can stop climbing the path K as soon as one proper point has been found. Otherwise climb up path K and continue with the predecessor branch node k_μ of k_ν if no proper point is stored in the nodes between k_ν and k_μ. After processing K we output the "best" proper point or the *nil*-point.

To perform the operation *AllProper* we walk down the path K as described above. During this walk it is sufficient to examine the points stored along K and the right subtrees rooted by branch nodes (see Lemma 1.2):
Let k_ν be a branch node. Then for all points q stored in $RST(k_\nu)$ we have $\mathcal{K}_2(p_0) < \mathcal{K}_2(q)$ by Lemma 1.1. The heap property of the CQPST with respect to $\mathcal{K}_1$ guarantees that all points in the subtree rooted by a node which stores a point with $\mathcal{K}_1$-value smaller than $\mathcal{K}_1(p_0)$ are non-proper. Therefore all proper points in $RST(k_\nu)$ can be reported by walking down paths starting in the root node of $RST(k_\nu)$, stopping as soon as the first point is found for which no proper point is contained in its subtrees. The paths may be constructed by breadth-first or depth-first search. The proper points found during this process are returned.

A straightforward and easy analysis of the CQPST shows that *insert*, *delete*, and *MinProper* can be performed in $O(\log n)$ time, and that *AllProper* requires $O(\log n + k)$ time where k denotes the number of proper points reported. The CQPST can be implemented such that its space requirement is linear in the number of points it stores.

2 Applying the CQPST to the ANFN problem

In this section we present an application of the CQPST to the *all-nearest-foreign-neighbors* (ANFN) problem:

> Given a finite set S of points in the plane $I\!R^2$, $|S| = n$, $S = \cup_{i=1}^{N} S_i$ with $S_i \cap S_j = \emptyset$ for $i, j \in \{1, \ldots, N\}$, $i \neq j$. Determine for each point $p \in S_i$ a nearest neighbor in $S \setminus S_i$.

We reformulate the problem in an intuitive way: Let us *color* each point set with a unique color and let $c(p)$ denote the color of a point $p \in S$. Now we have to find for each point a nearest neighbor with a different color. Since algorithms which solve the ANFN problem also solve the all-nearest-neighbors problem by choosing configurations that do not contain two points with the same color, $\Omega(n \log n)$ is a lower bound for the ANFN problem.
Distances between points are measured with respect to an arbitrary L^t-metric. For two points $p, q \in I\!R^2$ their L^t-distance

(Minkowski distance) is given by $d_t(p,q) := (|p.x - q.x|^t + |p.y - q.y|^t)^{\frac{1}{t}}$, $1 \le t < \infty$, and $d_\infty(p,q) := \max\{|p.x - q.x|, |p.y - q.y|\}$. For the L^1- and $L^\infty-$ metrics [3] presents an algorithm for solving the ANFN-problem in optimal $O(n \log n)$ time.

The ANFN problem is solved by first ***distributing*** the points: Each color set S_i is assigned a ***candidate set*** C_i, and the points are distributed among the candidate sets such that the total number of points contained in the candidate sets is $8n$. After the distribution a nearest foreign neigbor of a point $p \in S_i$ can be found in the candidate set C_i. Hence we have to solve a bichromatic ANFN-problem for each set S_i and its candidate set C_i; all points in the candidate sets loose their original colors and are painted in a color different from the set's color. [1] presents the first distribution algorithm for the ANFN problem with respect to the L^2-metric making use of several L^2-Voronoi diagrams.

The algorithm presented in this paper uses the CQPST. In the final phase of our algorithm, i.e. solving the bichromatic ANFN problem for each set S_i and its candidate set C_i we apply the algorithm in [5] which also works for arbitrary L^t-metrics.

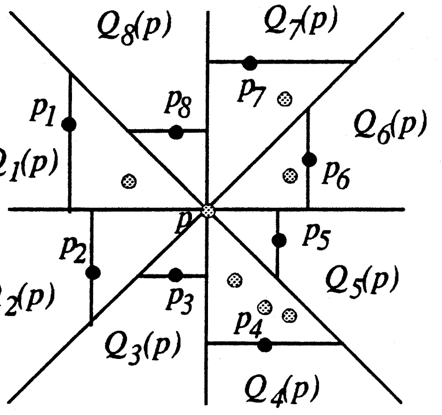

Figure 3: Subdivision with L^∞-nearest-foreign neighbors p_i

Fix an arbitrary point $p \in S$ and subdivide the plane into eight octants $Q_1(p), \dots, Q_8(p)$ (see Figure 3). For each octant we determine a L^∞-nearest-foreign neighbor of p, these points are denoted by $p_1, \dots, p_8$. It suffices to show how to find a point p_1 in the octant $Q_1(p)$. The points $p_2, \dots, p_8$ for the other octants are determined analogously.

We could choose $\mathcal{K}_1 = -(x+y)$ and $\mathcal{K}_2 = y$ which describe for $\mathcal{K}_1 > -(p.x + p.y)$ and $\mathcal{K}_2 > p.y$ the octant $Q_1(p)$. However, we do not search for a nearest foreign neighbor of p in $Q_1(p)$ with respect to $\mathcal{K}_1$ or $\mathcal{K}_2$ as it is supported by the CQPST but with respect to the x-coordinate. Hence the problem cannot be solved directly. To find a point $p_1 \in Q_1(p)$ we perform a top-down sweep and insert the π-tansformed points into the CQPST where π is the following transformation: $\pi : (x,y) \longrightarrow (x, x+y)$. It is easy to verify that a point q is contained in $Q_1(p)$ iff $\pi(q)$ is contained in $Q_2(p) \cup Q_3(p)$. We therefore choose $\mathcal{K}_1 = -x$ and $\mathcal{K}_2 = -y$. Clearly, the L^∞-nearest-foreign neighbor p_1 of p in $Q_1(p)$ has already been inserted into the CQPST when the sweep-line encounters p. Then $MinProper(p)$ returns the L^∞-nearest-foreign neighbor p_1 since the x-coordinates of the points are invariant under π. The following Lemma implies that if a point $q \in S$ has p as nearest foreign neighbor then $c(q) = c(p_i)$ for at least one $i \in \{1, \dots, 8\}$.

Lemma 2.1. *Let $p_1 \in S$ be a L^∞-nearest-foreign neighbor of $p \in S$ contained in $Q_1(p)$. Then for each point $r \in Q_1(p)$ with $c(p) \neq c(r)$ and $c(p_1) \neq c(r)$ we have*

$$d_t(r, p_1) < d_t(r, p)$$

The proof is easy, it uses the same argument as the proof of Theorem 2 in [7].

Hence we insert the point p into the candidate sets corresponding to the points $p_1, \dots, p_8$. Lemma 2.1 then implies that for each point $p \in S$ a nearest foreign neighbor can be found in the candidate set of its color $c(p)$. This shows that after the final phase of the algorithm, i.e. solving the bichromatic ANFN problems for the sets S_i and their assigned candidate sets C_i, for each point a nearest foreign neighbor has been determined correctly.

The analysis of the algorithm is straightforward. In each sweep each point is inserted into a CQPST exactly once. For each insertion event we perform two *MinProper*-queries which cost $O(\log n)$ time. Hence the total cost for the sweeps to distribute the points among the candidate sets sums up to $O(n \log n)$. Applying the plane-sweep algorithm [5] to each of the pairs (S_i, C_i) $(i = 1, \dots, m)$ costs $O(n \log n)$ time in total. Since a CQPST requires linear space this is also true for our algorithm.

References

1 A.Aggarwal, H.Edelsbrunner, P.Raghavan and P.Tiwari: Optimal time bounds for some proximity problems in the plane, *Information Processing Letters* 42, 55-60 (1992).

2 Th. H. Cormen, Ch. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, MIT Press, 1989.

3 Th. Graf, K. Hinrichs: Algorithms on colored point sets, *Proceedings of the 5th Canadian Conference on Computational Geometry CCCG '93*, (1993).

4 L.J. Guibas and R. Sedgewick: A Dichromatic Framework for Balanced trees, *Proceedings of the 19th IEEE Sympsoium on Foundation of Computer Science*, 8-21 (1978).

5 K.Hinrichs, J.Nievergelt, P.Schorn: An all-round sweep algorithm for 2-dimensional nearest-neighbor problems, *Acta Informatica*, 29(4), 383-394 (1992).

6 Ch.Icking, R.Klein, Th.Ottmann: Priority search trees in secondary memory, in H. Göttler und H.J. Schneider (eds.), *Graphtheoretic Concepts in Computer Science* (WG '87), *Lecture Notes in Computer Science* 314, Springer-Verlag, New York, 84-93, (1987).

7 J.Katajainen: The region approach for computing relative neighborhood graphs in the L_p metric, *Computing* 40, 147-161 (1988).

8 D.T. Lee and C.K. Wong: Voronoi diagrams in L_1 (L_∞) metrics with 2-dimensional storage applications, *SIAM Journal on Computing* 9(1), 200-211 (1980).

9 E.M.McCreight: Priority Search Trees, *SIAM J.Comput.*, 14,**2**, (1985).

10 H.J.Olivie: A new class of balanced search trees: Half-balanced binary search trees, *R.A.I.R.O. Informatique Theorique* 16, 51-71, (1982).

11 Th. Ottmann, D. Wood: *Updating Binary Trees with Constant Linkage Cost*, Research Report CS-89-45, Data Structuring Group, University of Waterloo (1989).

12 Th. Ottmann and P. Widmayer: *Algorithmen und Datenstrukturen*, Reihe Informatik, Band 70, BI Wissenschaftsverlag, Mannheim, 1993.

13 F.P.Preparata and M.I.Shamos, *Computational Geometry*, 3rd pr., Springer-Verlag, New York, 1985.

14 M.Shamos, D.Hoey: Closest-point problems, *16th Annual IEEE Symposium on Foundation of Computer Science*, 151-162 (1975).

15 R.E. Tarjan, *Data Structures and Network Algorithms*, Society for Industrial Applied Mathematics, 1983.

16 P.Vaidya: An $O(n \log n)$ algorithm for the all-nearest-neighbours-problem, *Discrete & Computational Geometry* 4, 101-115 (1989).