Dynamic algorithms for approximate neighbor searching

(Extended abstract)

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

1 Introduction

We consider the dynamic problems of computing

- approximate nearest neighbor,
- approximate k-nearest neighbor,
- approximate range searching,
- approximate furthest neighbor and
- approximate diameter.

1.1 Approximate nearest neighbor problem

The nearest neighbor searching is one of the fundamental problems in computational geometry. We are given a set S of n points in $\mathbf{R^d}$, $d \geq 2$, and a distance metric $L_t, 1 \leq t \leq \infty$. It is assumed that the dimension d is a constant independent of n. Each point p is given as a d-tuple of real numbers $(p_1, \ldots, p_d)$. Let $dist(p, q)$ denote the distance between points p and q.

Definition 1.1 Given any $\varepsilon > 0$ and any query point $q \in \mathbf{R^d}$, a point $p \in S$ is a $(1+\varepsilon)$-*approximate nearest neighbor of* q if, for any $r \in S \setminus \{q\}$, $dist(p, q) \leq (1+\varepsilon) dist(r, q)$.

Kapoor and Smid [14] presented the dynamic data structure of size $O(n \log^{d-1} n)$ with an amortized update time $O(\log^{d-1} n \log\log n)$ and query time $O(\log^{d-1} n \log\log n)$. In [7] the author gave the data structure of size $O(n \log^{d-2} n)$ with query and update times of $O(\log^{d+1} n \log\log n)$. Arya et al. [4] used the box-decomposition tree (with midpoint decomposition) and the topology trees of Frederickson [13] to solve the approximate nearest neighbor problem. This data structure has size $O(n)$ and update time of $O(\log n)$. The query time is $O((1+1/\varepsilon)^d \log n)$.

In Section 3, we present a dynamic algorithm that computes an approximate neighbor in $O(\log n + 1/\varepsilon^{d-1})$ time. This algorithm is based on the fair split tree. To maintain the fair split tree we apply the dynamic trees of Sleator and Tarjan [16]. These trees have linear space and logarithmic update time.

1.2 Approximate range searching problem

Arya and Mount [5] introduced the approximate range searching problem and showed that the approximate queries can be answered in $O(\log n + 1/\varepsilon^d)$ time (for convex ranges time is $O(\log n + 1/\varepsilon^{d-1})$). In Section 4, we extend these results to the dynamic version of the problem.

1.3 Approximate furthest neighbor problem

The situation with the furthest neighbor searching is similar to that with the nearest neighbor searching. In planar case the static version of the problem can be solved optimally by using

furthest-neighbor Voronoi diagram [15]. In planar case and dynamic version of the problem, Agarwal et al.[1] obtained a data structure of size $O(n^{1+\varepsilon})$, for any $\varepsilon > 0$, so that points can be inserted into or deleted from S in time $O(n^{\varepsilon})$ per update, and a furthest-neighbor query, under any L_t-metric, can be answered in $O(\log n)$ time. It is natural to consider the *approximate furthest-neighbor problem.*

Definition 1.2 Given any $\varepsilon > 0$ and any query point $q \in \mathbf{R}^d$, a point $p \in S$ is a $(1+\varepsilon)$-*approximate furthest neighbor of* q if, for any $r \in S$, $dist(r,q) \leq (1+\varepsilon)dist(p,q)$.

Agarwal et al. [2] proposed an algorithm that computes, for each point $p \in S$, approximate furthest neighbor in $O(n\varepsilon^{(1-d)/2})$ time.

In Section 5 we modify the fair split tree and give an dynamic algorithm for finding $(1+\varepsilon)$-approximate furthest neighbor in $O(1/\varepsilon^{d-1})$ time. Note that the query time is independent of n.

1.4 Approximate diameter problem

The *diameter* of S is defined as $diam(S) = \max\{dist(p,q) : p,q \in S\}$. It is well known that the diameter of n-point set in plane can be found in $O(n \log n)$ time. Using parametric searching, Chazelle et al. [12] showed that the diameter in 3-space can be computed in $O(n^{1+\gamma})$ time, for any $\gamma > 0$.

In Section 6, we consider the dynamic version of the problem and show that $(1+\varepsilon)$-approximate diameter can be computed in $O((1+1/\varepsilon)^{2(d-1)})$ time.

2 The fair split tree

The fair split tree is a hierarchical subdivision of the space into boxes. We define a box to be the product $[a_1, a_1') \times \ldots \times [a_d, a_d')$ of d semiclosed intervals. 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 *d-cube.* The cubes are useful in some proximity algorithms (for example, all-nearest-neighbors algorithm of Vaidya [17]). Unfortunately we cannot directly use the cubes in subdivision of space for dynamic problem, because a split of cube by a hyperplane $x_i = const$ does not give cubes. Another way is the using of the almost cubical boxes (or c-boxes for brevity) [8] and a fair split [9, 10, 11] or an almost middle cut [8]. The almost middle cut is similar to the fair split (but there is the difference of the definitions). In this paper, for the split of boxes, we shall use the definition as in [8] but we shall call its the *fair split.*

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 [9, 10, 11] and the almost middle cut [8] 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 $\frac{\sqrt{5}+1}{2} \approx 1.62$.

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

Definition 2.2 Let $B = [a_1, a_1') \times \ldots \times [a_d, a_d')$ 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 $x_i = c_i$ is *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 d-dimensional boxes. The box A is said to be a *s-sub-box* of B if A can be constructed from B by applying a (possibly empty) sequence of fair cuts. We shall write $B \rightsquigarrow A$. For $d = 1$, we shall say that A is *s-sub-interval* of B.

The second definition of s-sub-interval [8] can be generalized.

Definition 2.4 Let $[a, a')$ and $[b, b')$ be intervals in $\mathbf{R}$. Let $[a, a')$ is the sub-interval of $[b, b')$, i.e. $b \leq a \leq a' \leq b'$. The interval $[a, a')$ is called *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 the sequence of fair cuts for two boxes A and B if $B \rightsquigarrow A$. The following Theorem gives the condition for separator when definitions 2.3 and 2.4 are equivalent.

Theorem 2.5 *The definitions 2.3 and 2.4 define the same relation of s-sub-interval if and only if the separator is at least Golden Ratio, i.e* $s \geq \frac{\sqrt{5}+1}{2} \approx 1.62$

The dependence of the constant factors in the query answering efficiency is $O((s+1)^d)$. The decreasing of the separator reduces these factors.

Definition 2.6 Let B be a box with sides $s_1, \ldots, s_k$. The box B is said to be a *c-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 the 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 c-boxes.

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

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

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

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

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 omit the description of the dynamic tree for brevity. It can be found in [6, 16, 8].

3 The approximate nearest neighbor queries

To find an approximate nearest neighbor we shall use dynamic tree. (It is easy to see that the depth of fair split tree can be linear in the worst case.) We apply the technique similar to the finding the sets E_p and $A(v)$ [8]. We give two algorithms. The code of the first algorithm is simple but the second algorithm has better computational complexity.

Both algorithms use a set V. An element of V is a node of the fair split tree or a node of a path tree. With each node $v \in V$ we associate a domain $D(v) \subset \mathbf{R^d}$. To define a domain $D(v)$ we consider three cases.

Case 1. v is a node of fair split tree. $D(v) = SB(v)$.

In the next two cases v is a node of a path tree PT.

Case 2. v is a leaf of a path tree. v corresponds to a node v' of T. We replace v in V with a node of T which is determined as follows. Note that v' is an internal node of T. Let u and w be sons of v'. If u and w are linked to v' by dashed edges then we replace v with v'. If the edge (v', u) is dashed and the edge $(v'w)$ is solid then we replace v with u.

Case 3. v is an internal node of a path tree. The node v covers nodes $v_1, \ldots, v_k$ of solid path where $parent(v_{i+1}) = v_i$ for $i = 1, \ldots, k-1$. If v_k is the bottommost node of path tree then $D(v) = SB(v_1)$ ($v_1 = btail(v)$). Otherwise v_k has a son v_{k+1} linked to it by a solid edge. The domain $D(v) = SB(v_1) \setminus SB(v_{k+1})$.

To process domains $D(v)$ for internal nodes of path trees we store two boxes $B_{out}(v)$ and $B_{in}(v)$ (possibly empty) such that $D(v) = B_{out}(v) \setminus B_{in}(v)$. This information allows us, for a node $v \in V$,

- compute $d_{min}(q, D(v))$ and $d_{max}(q, D(v))$ in $O(d)$ time
- compute $D(v_1)$ and $D(v_2)$ where v_1 and v_2 appear when we traverse the node v.

The variable R contains an upper bound of the distance from query point q to $(1+\varepsilon)$-approximate nearest neighbor. In fact this bound is $R = \min\{d_{max}(q, D(v)) \mid v \text{ is traversed in search step}\}$. Let v_R be a node such that $R = d_{max}(q, D(v_R))$. The procedure *refresh_R* computes this node and R after updates of V.

procedure $refresh\_R(v)$
  $R' := d_{max}(q, D(v))$
  if $R > R'$ **then** $R := R'$; $v_R := v$ **fi**
end $refresh\_R$;

Now we describe the first algorithm. Initially $V = \{v_{root}\}$, $v_R = v_{root}$ and $R = d_{max}(q, D(v_R))$. We repeat the search step while $V \neq \emptyset$.

SEARCH STEP. Take any $v \in V$. Remove v from V. If $d_{min}(q, D(v)) > \frac{R}{1+\varepsilon}$ then return (start new search step). If v is a leaf of T then return. If v is a leaf of path tree then we replace it with a node of T (see Case 2 above). If v is an internal node of T and v is linked to sons u and w by dashed edges then add u, w to V. If v is a topmost node of a solid path that contains more than one node then add sons u, w of $pt\_root(v)$ (in the path tree) to V. If v is an internal node of path tree then add sons u, w of v to V. $Refresh\_R(u)$ and $Refresh\_R(w)$.

After all search steps ($V = \emptyset$) any point in $D(v_R)$ is the $(1+\varepsilon)$-approximate nearest neighbor and can be found in $O(\log n)$ time. It is clear that the algorithm is correct. One can prove that the number of visited nodes is $O((\log n)/\varepsilon^{d-1})$.

The second algorithm has three phases. In the first phase, we compute $(1+\varepsilon_0)$-approximate distance from query point to nearest neighbor. To do this, we apply the first algorithm. ε_0 is a constant and we assign $\varepsilon_0 = 1$.

Our goal in the second phase is to obtain the set V of nodes of T such that

- for distinct nodes u and v, $B(u) \cap B(v) = \emptyset$
- the boxes $B(v), v \in V$ contain points of S that are within distance at most R, i.e. $\{p \in S, dist(p,q) \leq R\} \subset \cup_{v \in V} B(v)$.
- for a node $v \in V$, $diam(B(v)) < R$ and (if v has a parent) $diam(B(parent(v))) \geq R$.

To construct such a set of nodes we apply the search that is similar to the first algorithm.

In the third phase, we compute an $(1+\varepsilon)$-approximate nearest neighbor of query point. The algorithm has $O(\log(1+\varepsilon))$ iterations. At the beginning of iteration

- compute $R = \min_{v \in V} d_{max}(q, D(v))$. Note that $v \in T$ and computing d_{max} is simple.
- compute $r = \min_{v \in V} d_{min}(q, D(v))$.
- compute $d_1 = \max_{v \in V} diam(SB(v))$.
- remove v from V if $d_{min}(q, D(v)) > R$.

If $R \leq (1+\varepsilon)r$ then any point in $D(v).d_{min}(q, D(v)) = r$ is the $(1+\varepsilon)$-approximate nearest neighbor and can be found in $O(\log n)$ time. Set $U = \emptyset$. We repeat the search step while $V \neq \emptyset$.

SEARCH STEP. Take any $v \in V$. Remove v from V. If v is a leaf of T or $d_{min}(q, SB(v)) > R$ then return. If $diam(SB(v)) < d_1/2$ then add v to U and return. Add sons u, w of v to V and $Refresh\_R(u)$ and $Refresh\_R(w)$.

After the iteration we set $V = U$. Note that each iteration decreases maximal diameter d_1 of $SB(v), v \in V$ at least to $d_1/2$. The first and second phases take $O(\log n)$ time. The third phase without final finding of the $(1+\varepsilon)$-approximate nearest neighbor (in Section 5 it is shown that the sampling a point can be done in $O(1)$ time) takes $O((1+1/\varepsilon)^{d-1})$ time.

Theorem 3.1 *Using the fair split tree and dynamic tree, for query point q, any $\varepsilon > 0$ and metric L_t, $(1+\varepsilon)$-approximate nearest neighbor query can be answered in $O((1+1/\varepsilon)^{d-1} + \log n)$ time.*

3.1 The approximate k-nearest neighbor queries

Consider the problem of computing approximations to the k nearest neighbors of a query point.

Definition 3.2 Given any $\varepsilon > 0$ and any query point $q \in \mathbf{R^d}$, a point $p \in S$ is a $(1+\varepsilon)$-*approximate k-th nearest neighbor of q* if, for a true k-th nearest neighbor of q, $\frac{dist(p,q)}{dist(r,q)} \leq 1+\varepsilon$.

We can use the dynamic algorithm for approximate nearest neighbor searching. Let $ann(q, \varepsilon, t)$ be the function that returns $(1+\varepsilon)$-approximate nearest neighbor of query point q under L_t-metric. The procedure $aknn(q, \varepsilon, t)$ returns a list of points $p_1, \ldots, p_k$ that is an answer for $(1+\varepsilon)$-approximate k-nearest neighbor query.

procedure $aknn(q, \varepsilon, t)$
  for $i := 1, \ldots, k$
    $p_i := ann(q, \varepsilon, t)$
    $delete(p_i)$ ($*$ from S $*$)
  rof
  for $i := 1, \ldots, k$
    $insert(p_i)$ ($*$ to S $*$)
  rof
end $aknn$;

It is clear that the running time of the algorithm is $O(k((1+1/\varepsilon)^{d-1} + \log n))$. In fact $(1+\varepsilon)$-

approximate k-nearest neighbor problem can be solved $O(k + (1 + 1/\varepsilon)^{d-1} + \log n)$ time but the space restriction does not allow us to describe such algorithm.

4 The approximate range searching

We assume that the points have been assigned weights. In precise range searching, given any query range Q, we have to compute the accumulated weight of the points in $S \cap Q$, $weight(S \cap Q)$, under some commutative semigroup.

Definition 4.1 Given any $\varepsilon > 0$ and any query range Q of diameter d, define Q^- to be the locus of points whose distance from a point exterior to Q is at least $d\varepsilon$, and Q^+ to be the locus of points whose distance from a point interior to Q is at most $d\varepsilon$. Define a *legal answer* to an $(1+\varepsilon)$-approximate range query to be $weight(S')$ for any subset S' such that $S \cap Q^- \subset S' \subset S \cap Q^+$.

We can adapt the algorithm of Arya and Mount [5] to fair split tree and dynamic tree. The cells $cell(v)$ correspond to the domains $D(v)$. With each node of path trees we shall store $weight(v) = weight(D(v))$. We can maintain weights of nodes in $O(\log n)$ time per update of S.

Theorem 4.2 *Using the fair split tree and modified dynamic tree, for a spherical query range, any $\varepsilon > 0$ and metric L_t, $(1 + \varepsilon)$-approximate range count can be computed in $O((1 + 1/\varepsilon)^d)$ time.*

5 The approximate furthest neighbor queries

In this Section we shall give an algorithm for approximate furthest neighbor searching. The algorithm has simple code. The algorithm uses a set V of nodes of T. The variable r contains a lower bound for distance to $(1 + \varepsilon)$-approximate furthest neighbor and the box $B(v_r)$ contains it. Initialization: $V = \{v_{root}\}$, $r = d_{min}(q, SB(v_{root}))$, and $v_r = v_{root}$.

The procedure $Refresh\_r()$ is analog of $Refresh\_R()$ in Section 3.

procedure $refresh\_r(v)$
  $r' := d_{min}(q, SB(v))$
  if $r < r'$ **then** $r := r'$; $v_r := v$ **fi**
end $refresh\_r$;

We apply the *scaling technique* of phase 3 of approximate nearest neighbor algorithm in Section 3. The algorithm has $O(lg(1 + 1/\varepsilon))$ iterations. At the beginning of iteration

- compute $d_1 = \max_{v \in V} diam(SB(v))$.
- set $U = \emptyset$.

We carry out the search step while the set V is non-empty.

SEARCH STEP. Take any node $v \in V$. Remove v from V. If $d_{max}(q, SB(v)) \leq (1 + \varepsilon)r$ then return. If $diam(SB(v)) \leq d_1/2$ then add v to U and return. Let u and w be sons of v (in T). Add u and w to V. $Refresh\_r(u)$ and $Refresh\_r(w)$.

After iteration we assign $V = U$.

This algorithm without final finding of the $(1+\varepsilon)$-approximate furthest neighbor takes $O((1 + 1/\varepsilon)^{d-1})$ time. We can compute the approximate furthest neighbor using the point location. However the point location has $O(\log n)$ time in worst case. We modify the fair split tree such that the final finding can be performed in constant time. We add the pointer $point()$ to nodes of V. For a node $v \in V$, $point(v)$ is a point in $S \cap B(v)$ if the edge $(v, parent(v))$ of $\Delta(T)$ is dashed. We can maintain this pointer in $O(\log n)$ time per update of S. To find any point in $B(v_r)$ we choose a son u of v such that the edge (u, v) is dashed (if v is a leaf then the finding is trivial). $point(u)$ points to required point of S.

6 The approximate diameter

In this Section we shall give an algorithm for approximate diameter searching. The algorithm uses a set V of nodes of T. We apply the scaling technique. The algorithm has $O(lg(1 + 1/\varepsilon))$ iterations. At the beginning of iteration

- compute $r = \max_{u,v \in V} d_{min}(SB(u), SB(v))$ and nodes u_r and v_r such that $r = d_{min}(SB(u_r), SB(v_r))$.

- for any node $v \in V$, compute $R_v = \max_{u \in V} d_{max}(SB(u), SB(v))$. If $R_v \leq (1+\varepsilon)r$ then remove node v from V.
- If $V = \emptyset$ then $(1+\varepsilon)$-approximate diameter is $dist(p,q)$ where $p \in S \cap SB(u_r)$ and $q \in S \cap SB(v_r)$ and the algorithm stops.
- compute $d_1 = \max_{v \in V} diam(SB(v))$.
- set $U = \emptyset$.

We carry out the scaling step while the set V is non-empty.

SEARCHING STEP. Take any node $v \in V$. Remove v from V. If $diam(SB(v)) \leq d_1/2$ then add v to U and return. Let u and w be sons of v (in T). Add u and w to V.

After iteration we assign $V = U$.

As in the preceding Section we use pointers $point()$ to determine the pair (p,q) that gives approximate diameter. The number of nodes in V is at most $O((1+1/\varepsilon)^{d-1})$. The running time of an iteration (i.e. searching steps) is $O((1+1/\varepsilon)^{d-1})$. The part before an iteration (i.e. computing r, R_v) takes $O((1+1/\varepsilon)^{2(d-1)})$ time. One can prove that the running time of the algorithm is within a constant factor of this time.

References

[1] P. K. Agarwal, A. Efrat, and M. Sharir. *Vertical Decomposition of Shallow Levels in 3-Dimensional Arrangements and Its Applications.* Proc. 11th Annual ACM Symp. Comput. Geom., 1995, pp. 39–50.

[2] P. K. Agarwal, J. Matoušek, and S. Suri. *Farthest Neighbors, Maximum Spanning Trees and Related Problems in Higher Dimensions.* Computational Geometry: Theory and Applications, 1992, pp. 189–201.

[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. *An Optimal Algorithm for Approximate Nearest-Neighbor Searching.* Proc. 5th Annual Symposium on Discrete Algorithms, 1994, pp. 573–582.

[4] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. *An Optimal Algorithm for Approximate Nearest-Neighbor Searching.* (revised version), 1994.

[5] S. Arya and D. M. Mount. *Approximate Range Searching.* Proc. 11th Annual ACM Symp. Comput. Geom., 1995, pp. 172–181.

[6] S. W. Bent, D. D. Sleator and R.E. Tarjan *Biased Search Trees.* SIAM Journal of Computing, 1985, 14, pp. 545-568

[7] S. N. Bespamyatnikh. *The Region Approach for Some Dynamic Closest-Point Problems.* Proc. 6th Canadian Conf. Comput. Geom., 1994.

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

[9] 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.* Proc. 24th Annual AMC Symposium on the Theory of Computing, 1992, pp. 546–556.

[10] P. B. Callahan and S. R. Kosaraju. *Algorithms for Dynamic Closest Pair and n-Body Potential Fields.* Proc. 6th Ann. Symp. on Discrete Algorithms, 1995.

[11] P. B. Callahan, M. T. Goodrich and K. Ramaiyer. *Topology B-Trees and Their Applications.* WADS conference.

[12] B. Chazelle, H. Edelsbrunner, and L. J. Guibas. *Diameter, Width, Closest Line Pair, and Parametric Searching.* Proc. 8th Annual ACM Symp. Comput. Geom., 1992, pp. 120–129.

[13] G. N. Frederickson. *A Data Structure for Dynamically Maintaining Rooted Trees.* Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms, 1993, pp. 175-194.

[14] 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.

[15] D. T. Lee. *Farthest neighbor Voronoi diagrams and applications.* Tech. Rept. No. 80-11-FC-04, Dpt. EE/CS, Northwestern University,1980.

[16] D. D. Sleator and R. E. Tarjan. *A Data Structure for Dynamic Trees.* Journal of Computer and System Sciences, 26, 1983.

[17] P. M. Vaidya. *An $O(n \log n)$ Algorithm for All-Nearest-Neighbors Problem.* Discrete Comput. Geom., 1989, pp. 101–115