A Minimalist's Implementation of an Approximate Nearest Neighbor Algorithm in Fixed Dimensions*

Timothy M. Chan†

May 9, 2006

Abstract

We consider the standard problem of approximate nearest neighbor search, for a given set of n points with integer coordinates in a constant-dimensional Euclidean space. We describe a simple implementation of a randomized algorithm that guarantees $O(\log n)$ expected query time and $O(n \log n)$ preprocessing time. The entire C++ code is under 100 lines long and requires no extra space other than the input array. The algorithm can easily be made dynamic as well.

1 Introduction

Approximate nearest neighbor search [1, 18] is one of the most well-researched problems in computer science. The goal of this article is to describe a *provably efficient* method when the input is a point set in Euclidean space in a *low dimension* d (like 2 or 3). This fixed-dimensional Euclidean case is arguably the most natural case to consider (although high-dimensional and more general metric-space settings have also received much attention [6, 11]). Methods with provable performance guarantees (in contrast to, say, traditional k-d trees and quadtrees [16]) are appealing in that analyses hold for all input point sets, without any assumptions about their distributions.

For $d = 2$, techniques from computational geometry (Voronoi diagrams and point location methods) [15] yield data structures for *exact* nearest neighbor search, with $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time, matching the standard 1-d bounds (which are optimal among comparison-based methods). However, these techniques are fairly complicated and do not work well when the dimension exceeds two.

Arya *et al.* [2] were the first to obtain an algorithm for *approximate* nearest neighbor search with $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time in any fixed dimension. Here, the point returned by the algorithm is guaranteed to have distance within a factor of $1 + \varepsilon$ from the minimum distance to the query point, for any fixed $\varepsilon > 0$. (Constant factors depend on ε and d.) The paper is somewhat lengthy—to guarantee good worst-case query times, the challenge is to find a hierarchical structure (in Arya *et al.*'s case, the *BBD tree*) which is "balanced" in both the combinatorial and geometric sense. Since Arya *et al.*'s work, other structures with additional functionalities (such as *BAR trees* [8] and *approximate Voronoi diagrams* [10, 3]) have been proposed (which are not necessarily simpler than the original BBD trees).

*This work is supported by NSERC.

†School of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1, Canada, `tmchan@uwaterloo.ca`.

In this article, we are interested in the *simplest* implementation of an approximate nearest neighbor algorithm that guarantees $O(n \log n)$ preprocessing time, $O(n)$ space, and $O(\log n)$ query time in any fixed dimension. In an earlier 2-page note, the author [5] has already noticed that a "shift-shuffle-and-sort" approach (inspired by some previous work [4, 13]) can lead to a simple algorithm with the desired performance. Here, we provide a more detailed account of one further simplified version of this approach. (Specifically, the simplification uses one random shift instead of multiple shifts; the new description also avoids the use of certain unspecified constants and results in a potentially faster implementation in practice.)

Our new algorithm has the following attractive features:

- Our algorithm is very easy to implement. The complete code fits in 2 pages (see the appendix).
- Our algorithm is *in-place*, i.e., it requires no extra space (in particular, no pointers) except for the input array, which stores an appropriate permutation of the points. (It might be possible to make some of the previous data structures in-place, but this would require extra effort.) In a way, our method is analogous to, and is almost as simple as, the standard 1-d searching method: preprocess by sorting the input array, and answer queries by performing a binary search in the sorted array.
- Our algorithm can be easily made dynamic, supporting insertions and deletions in $O(\log n)$ time (this time, pointers are required). In contrast, dynamizing Arya *et al.*'s original method requires sophisticated techniques (namely, the use of Sleator and Tarjan's dynamic trees or Frederickson's topology trees). Eppstein, Goodrich, and Sun [9] have recently suggested the *skip quadtree* as a simple approximate nearest neighbor method that supports dynamic updates, but our approach is even simpler—we can directly invoke any standard data structure for 1-d search (for example, a red-black tree, AVL tree, treap, or any of its cousins [7]).

To be fair, we mention the following drawbacks:

- Our algorithm does not work in the real-RAM model. Instead we assume that each coordinate is an integer and fits in a word, and that common operations on words (specifically, bitwise-exclusive-or and most-significant-bit) are available. This assumption is common and is made, for instance, in an early version of the BBD tree [2] as well as the skip quadtree [9].
- Our algorithm is randomized. The $O(\log n)$ query time bound holds in expectation under an "oblivious" adversary assumption, namely, that query points are independent of the random choice(s) made by the preprocessing algorithm. This type of assumption is reasonable (and is also made, for example, in randomized analyses of hashing). We emphasize that the expected bound is not with respect to any input distribution and is valid for arbitrary point sets.
- Our primary goal is in simplicity of implementation, not actual speed in practice. Our implementation is reasonably efficient and is much faster than brute-force search in low dimensions, but is slower than, for example, Arya *et al.*'s fine-tuned code [14].

In the next section, we provide an entirely self-contained description and analysis of our algorithm.

2 A Shift-Shuffle-Sort Method ("SSS")

Preliminaries. Given a point $p \in \mathbb{N}^d$ whose coordinates $(p_{1w} \cdots p_{10},\ p_{2w} \cdots p_{20},\ \ldots,\ p_{dw} \cdots p_{d0})$ are written in binary, its *shuffle* refers to the number $p_{1w}p_{2w} \cdots p_{dw} \cdots p_{10}p_{20} \cdots p_{d0}$ written in binary. We write $p \preceq q$ iff the shuffle of p is less than or equal to the shuffle of q; the relation $\preceq$ is called the *shuffle order*.

To see the geometric significance of the shuffle order, we make the following definition: a *quadtree box* is a hypercube of the form $[k_1 2^i, (k_1+1)2^i) \times \cdots \times [k_d 2^i, (k_d+1)2^i)$ for some $i, k_1, \ldots, k_d \in \mathbb{N}$. These boxes form a tree structure (hence the name), where a quadtree box of side length 2^i can be partitioned into precisely 2^d quadtree subboxes of side length 2^{i-1}. A useful observation is that points inside a common quadtree box appear consecutively in the shuffle order—in other words, if $p_1 \preceq p_2 \preceq p_3$, then any quadtree box containing p_1 and p_3 must also contain p_2.

Let $B(p_1, p_2)$ denote the smallest quadtree cell containing two points p_1 and p_2. Given a point $p \in \mathbb{N}^d$ and a number s, let p^s denote the *shifted* point $p + (s, \ldots, s)$. Let $d(p, q)$ denote the (minimum) Euclidean distance between two points (or objects) p and q.

The algorithm. Our preprocessing algorithm is extremely simple. Given input points $p_1, \ldots, p_n \in \{0, \ldots, 2^w - 1\}^d$, we perform just one sorting step:

1. randomly pick a shift $s \in \{0, \ldots, 2^w - 1\}$
2. sort $p_1^s, \ldots, p_n^s$ according to the shuffle order $\preceq$

Our query algorithm is a modified binary search. Given a point $q \in \{0, \ldots, 2^w - 1\}^d$, we call the recursive procedure below, which finds a $(1+\varepsilon)$-factor approximate nearest neighbor of q in the subarray $p_a, \ldots, p_b$. Initially, $a = 1$ and $b = n$, and the global variable r is set to ∞. At the end, r contains the approximate minimum distance to q. (We can easily return the point that attains this distance.)

query$(q,\ a,\ b)$:

0. if $a > b$ then return
1. $m \leftarrow \lfloor (a+b)/2 \rfloor$
2. $r \leftarrow \min\{r, d(p_m, q)\}$
3. if $a = b$ or $d(q^s, B(p_a^s, p_b^s)) \geq \frac{r}{1+\varepsilon}$ then return
4. if $q^s \preceq p_m^s$ then {
5.   query$(q,\ a,\ m-1)$
6.   if $q^{s+\lceil r \rceil} \succeq p_m^s$ then query$(q,\ m+1,\ b)$

    }

7. else {
8.   query$(q,\ m+1,\ b)$
9.   if $q^{s-\lceil r \rceil} \preceq p_m^s$ then query$(q,\ a,\ m-1)$

    }

Correctness. In line 3 of the query algorithm, suppose that $d(q^s, B(p_a^s, p_b^s)) \geq \frac{r}{1+\varepsilon}$. Because $p_a^s, p_{a+1}^s, \ldots, p_b^s \in B(p_a^s, p_b^s)$, all points in the subarray $p_a, p_{a+1}, \ldots, p_b$ have distance at least $\frac{r}{1+\varepsilon}$ to q and thus need not be examined.

In line 6, suppose $q^{s+\lceil r \rceil} \preceq p_m^s$. Let $D_\infty(q, r)$ denote the box centered at q of side length $2r$. Because $q^{s+\lceil r \rceil}$ is the maximum point in the box $D_\infty(q^s, \lceil r \rceil)$ with respect to $\preceq$, we have $p_{m+1}^s, \ldots, p_b^s \notin D_\infty(q^s, \lceil r \rceil)$. So, all points in the subarray $p_{m+1}, \ldots, p_b$ have L_∞-distance (and, in particular, L_2-distance) at least r to q, and thus need not be examined.

Similarly, in line 7, if $q^{s-\lceil r \rceil} \succeq p_m^s$, then the subarray $p_a, \ldots, p_{m-1}$ need not be examined. The correctness of the query algorithm now follows.

Primitives. In order to implement the algorithm, we need two nonstandard primitives: given two points $p = (x_1, \ldots, x_d)$ and $q = (y_1, \ldots, y_d)$, (i) decide whether $p \preceq q$, and (ii) compute $B(p, q)$.

Let $x \operatorname{xor} y$ denote the bitwise exclusive-or of two natural numbers x and y. Let $\operatorname{msb}(x)$ denote the position of the most significant bit (i.e., leftmost 1-bit) of x. Note that the xor operation is commonly available, and msb can be computed using the logarithm function ($\operatorname{msb}(x) = \lfloor \log_2 x \rfloor$), or by converting x to floating point and extracting the exponent.

For (i), we find the index j with the largest $\operatorname{msb}(x_j \operatorname{xor} y_j)$; in case of ties, pick the smallest such index. Then $p \preceq q$ iff $x_j \leq y_j$. As noted earlier [5], we can actually avoid computing msb explicitly here, since $\operatorname{msb}(x) < \operatorname{msb}(y)$ iff x is smaller than both y and $x \operatorname{xor} y$.

For (ii), let $i = 1 + \max_{j=1,\ldots,d} \operatorname{msb}(x_j \operatorname{xor} y_j)$. Then $B(p, q) = [\lfloor x_1/2^i \rfloor 2^i, (\lfloor x_1/2^i \rfloor + 1)2^i) \times \cdots \times [\lfloor x_d/2^i \rfloor 2^i, (\lfloor x_d/2^i \rfloor + 1)2^i)$, which can be computed using shifts.

Analysis. Preprocessing takes $O(dn \log n)$ time by any optimal comparison-based sorting algorithm, since each comparison takes $O(d)$ time.

The query time is bounded by $O(d)$ times the number N of internal nodes in the recursion tree generated by the query algorithm. The tree obviously has $O(\log n)$ height. At an internal node, we say that the algorithm "visits" the quadtree box $B(p_a^s, p_b^s)$ for the corresponding indices a and b. Let p_* denote the exact nearest neighbor of q. Let i be such that $2^{i-1} \leq d(p_*, q) \leq 2^i$. Fix a value c and assume that the following condition $\mathcal{E}_c$ is true:

> $D_\infty(q^s, 2^i)$ is contained in a quadtree box of length 2^{i+c}, and $D_\infty(q^s, \sqrt{d}2^{i+c})$ is contained in a quadtree box of length 2^{i+2c}.

Then we have the following observations:

- The quadtree boxes at all but $O(\log n)$ nodes have side lengths at most 2^{i+2c}.

  Proof: Say $p_k^s \preceq q^s \preceq p_{k+1}^s$. When the first leaf in the tree is reached, both the distances $d(p_k, q)$ and $d(p_{k+1}, q)$ have been examined (in line 2). One of p_k^s or p_{k+1}^s is in between p_*^s and q^s with respect to $\preceq$, and must be in $B(p_*^s, q^s)$, which has side length at most 2^{i+c}. Thus, after the first leaf is reached, we have $r \leq r_0 := \sqrt{d}2^{i+c}$.

  Each quadtree box $B(p_a^s, p_b^s)$ visited must have $q^{s+\lceil r_0 \rceil} \succeq p_{a-1}^s$ or $q^{s-\lceil r_0 \rceil} \preceq p_{b+1}^s$ (because of the if conditions in lines 6 and 9). At each level of the tree, there are at most two nodes with $p_{a-1}^s \preceq q^{s+\lceil r_0 \rceil} \preceq p_{b+1}^s$, or $p_{a-1}^s \preceq q^{s-\lceil r_0 \rceil} \preceq p_{b+1}^s$. All other nodes have $q^{s-\lceil r_0 \rceil} \preceq p_{a-1}^s, p_{b+1}^s \preceq q^{s+\lceil r_0 \rceil}$, which implies that $B(p_a^s, p_b^s)$ has side length at most 2^{i+2c}.

- All quadtree boxes visited have side lengths at least $\Omega(\varepsilon/\sqrt{d})2^i$.

  Proof: At an internal node with quadtree box $B = B(p_a^s, p_b^s)$ of side length ℓ, we have the following inequalities (because of the if condition in line 3):

$$d(p_m, q) - \sqrt{d}\ell \;\le\; d(q^s, B) \;\le\; \frac{r}{1+\varepsilon} \;\le\; \frac{d(p_m, q)}{1+\varepsilon} \quad\Longrightarrow\quad \sqrt{d}\ell \ge \frac{\varepsilon d(p_m, q)}{1+\varepsilon}. \tag{1}$$

  Thus, $\ell = \Omega(\varepsilon/\sqrt{d}) d(p_m, q) \ge \Omega(\varepsilon/\sqrt{d})2^{i-1}$.

- The number of different quadtree boxes visited that have a given side length is $O(\sqrt{d}/\varepsilon)^d$.

  Proof: By (1), a quadtree box visited of side length ℓ must have distance at most $O(\sqrt{d}\ell/\varepsilon)$ from q. There are at most $O(\sqrt{d}/\varepsilon)^d$ such boxes.

- A quadtree box may be visited at most $O(2^d \log n)$ times.

  Proof: If $B = B(p_a^s, p_b^s) = B(p_{a'}^s, p_{b'}^s)$ with $p_a^s \preceq p_b^s \preceq p_{a'}^s \preceq p_{b'}^s$, then p_a^s and $p_{a'}^s$ must lie in different quadtree subboxes of B (because otherwise $B(p_a^s, p_b^s)$ would be strictly contained in B). Thus, B is visited at most 2^d times at each level of the tree.

Putting all these observations together, we conclude that if $\mathcal{E}_c$ holds,

$$N \;=\; \log\left(\frac{2^{i+2c}}{(\varepsilon/\sqrt{d})2^i}\right) \cdot O(\sqrt{d}/\varepsilon)^d \cdot 2^d \log n \;=\; O(\sqrt{d}/\varepsilon)^d (c + \log(1/\varepsilon)) \log n.$$

Now, if $\mathcal{E}_c$ does not hold, then one of the d coordinates of q^s must lie in the range $\{0, \pm 1, \ldots, \pm 2^i\}$ modulo 2^{i+c}, or $\{0, \pm 1, \ldots, \pm\lceil \sqrt{d}2^{i+c}\rceil\}$ modulo 2^{i+2c}. As s is random,

$$\Pr\{\overline{\mathcal{E}_c}\} \;=\; O\left(d\left(\frac{2^i}{2^{i+c}} + \frac{\sqrt{d}2^{i+c}}{2^{i+2c}}\right)\right) \;=\; O\left(\frac{d^{3/2}}{2^c}\right).$$

The expected running time is thus bounded by

$$\begin{aligned}
O(d)E[N] \;&=\; O(d)\sum_{c=1}^{\infty} E[N | \mathcal{E}_c \cap \overline{\mathcal{E}_{c-1}} \cap \cdots \cap \overline{\mathcal{E}_1}]\Pr\{\mathcal{E}_c \cap \overline{\mathcal{E}_{c-1}} \cap \cdots \cap \overline{\mathcal{E}_1}\} \\
&\le\; O(d)\sum_{c=1}^{\infty} E[N | \mathcal{E}_c \cap \overline{\mathcal{E}_{c-1}} \cap \cdots \cap \overline{\mathcal{E}_1}]\Pr\{\overline{\mathcal{E}_{c-1}}\} \\
&=\; O(\sqrt{d}/\varepsilon)^d \left(\sum_{c=1}^{\infty} \frac{c}{2^c} + \log(1/\varepsilon)\right) \log n \;=\; O(\sqrt{d}/\varepsilon)^d \log(1/\varepsilon) \log n.
\end{aligned}$$

3 Remarks

Implementation. The appendix gives an implementation in C++, using available operations for xor (`^`), msb (`frexp`), shifts, and square roots, as well as the `qsort` library function. Some quick experiments (run on a Sun Ultra 10) indicate that the code is reasonably fast in low dimensions. For example, for uniformly distributed input and query points in $\{0, \ldots, 2^{29}-1\}^d$ with $n = 300000$,

$d = 3$, and $\varepsilon = 0$, preprocessing takes 4.5 seconds and a query takes 0.0005 seconds; for $n = 300000$, $d = 10$, and $\varepsilon = 3$, preprocessing takes 10.5 seconds and a query takes 0.067 seconds. (Of course, our analysis provides no guarantees on the query time when $\varepsilon = 0$. Also, for $\varepsilon > 0$, as in Arya *et al.*'s implementation, the approximate neighbors returned tend to have factor better than $1+\varepsilon$.) We will omit a full experimental report, as Arya *et al.*'s k-d tree code [2, 14] achieves better query times in practice (though our preprocessing is often slightly faster).

Dynamic version. Our method is perhaps the easiest to dynamize. We just maintain the sorted list of points in a balanced search tree instead of an array. Each update requires $O(\log n)$ comparisons and thus takes $O(d \log n)$ time. The query algorithm requires only cosmetic changes and still runs in logarithmic time, since the tree height is $O(\log n)$.

The simplest option for the balanced search tree is perhaps Seidel and Aragon's randomized *treap* [17]. The resulting C++ code for dynamic approximate nearest neighbors is just 148 lines long.

Some variants. Some alternative expressions may replace $d(q^s, B(p_a^s, p_b^s))$ (in line 3) without affecting our analysis. For example, for a stronger bound that may yield slightly better results in practice, we can use the distance of q^s to the smallest *binary*-quadtree box containing p_a^s and p_b^s. Here, a binary-quadtree box refers to a hypercube of the form $[k_1 2^i, (k_1{+}1)2^i) \times \cdots \times [k_j 2^i, (k_j{+}1)2^i) \times [k_{j+1}2^{i+1}, (k_{j+1}+1)2^{i+1}) \times \cdots \times [k_d 2^{i+1}, (k_d+1)2^{i+1})$ for some $i, k_1, \ldots, k_d \in \mathbb{N}$ and $j \in \{1, \ldots, d\}$. Each each such box can be partitioned into two binary-quadtree subboxes.

Alternatively, we can use the weaker bound $d(p_m, q) - \Delta$ where Δ is some constant-factor upper bound of the diameter of $B(p_a^s, p_b^s)$. Though slower in practice, the advantage of this approach is that the msb operation can be completely avoided (since an estimate for Δ can be computed with only the xor operation).

The shuffle order may also be replaced with other similar orders induced by space-filling curves (like Hilbert curves) [12, 13]. This may possibly lead to fewer nodes visited in practice, though comparisons based on, say, the Hilbert curve are harder to implement using the commonly available word operations.

If shuffles can be explicitly computed, we can reduce the preprocessing time to $O(n \log n + dn)$ by known multi-key sorting methods, since we are sorting n values each of which fit in d words. Furthermore, as mentioned in the previous note [5], it is possible to beat the $n \log n$ preprocessing time and $\log n$ query time barriers in theory, by using known (complicated) algorithms for 1-d sorting and searching on the word RAM.

Our method works, of course, for other L_p-metrics. It can be made deterministic, by using multiple shifts as noted previously [4, 5], though this would increase the preprocessing time and space by a factor of $O(d)$.

Quadtree variants. By storing more information during preprocessing, we can avoid regenerating the same quadtree box $O(\log n)$ times and potentially improve the constant factors in the query time bound. The authors' previous note [5] mentioned one way, using priority search trees. Another way is to augment the sorted list with an explicit representation of the "compressed" binary quadtree, i.e., to add extra links between the subarray $p_a, \ldots, p_b$ and the two subarrays $p_a, \ldots, p_{m'}$ and $p_{m'}, \ldots, p_b$, where m' is the largest index such that the smallest binary quadtree box containing p_a and $p_{m'}$ does not contain p_b (which can be found by binary search). It is not difficult to maintain the links of the compressed binary quadtree in $O(d \log n)$ time per insertion and deletion, by doing searches in our sorted list. With this augmented data structure, we can obtain a query algorithm that takes $O(d \log n) + O(\sqrt{d}/\varepsilon)^d \log(1/\varepsilon)$ expected time, though the algorithm becomes more complicated (and thus will not described here).

Alternatively, we could just use the binary quadtree alone, without our sorted list. By our analysis, a simple top-down query algorithm takes $O(dh) + O(\sqrt{d}/\varepsilon)^d \log(1/\varepsilon)$ expected time, where $h \le dw$ is the height of the binary quadtree. Although h could be larger than $\log n$, the second term usually dominates in practice except for the lowest dimensions. Our analysis thus partially explains the empirical success of a direct quadtree approach.

References

[1] S. Arya and D. M. Mount. Computational geometry: proximity and location. In *Handbook of Data Structures and Applications*, Chapman & Hall/CRC, pages 63.1–63.22, 2005.

[2] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45:891–923, 1998.

[3] S. Arya, T. Malamatos, and D. M. Mount. Space-efficient approximate Voronoi diagrams. In *Proc. 34th ACM Sympos. Theory Comput.*, pages 721–730, 2002.

[4] T. M. Chan. Approximate nearest neighbor queries revisited. *Discrete Comput. Geom.*, 20:359–373, 1998.

[5] T. M. Chan. Closest-point problems simplified on the RAM. In *Proc. 13th ACM-SIAM Sympos. Discrete Algorithms*, pages 472–473, 2002.

[6] K. L. Clarkson. Nearest-neighbor searching and metric space dimensions. In *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice* (T. Darrell, P. Indyk, G. Shakhnarovich, and P. Viola, ed.), MIT Press, 2006.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* McGraw-Hill, 2nd ed., 2001.

[8] C. A. Duncan, M. T. Goodrich, and S. Kobourov. Balanced aspect ratio trees: combining the advantages of k-d trees and octrees. *J. Algorithms*, 38:303–333, 2001.

[9] D. Eppstein, M. T. Goodrich, and J. Z. Sun. The skip quadtree: a simple dynamic data structure for multidimensional data. In *Proc. 21st ACM Sympos. Comput. Geom.*, pages 296–305, 2005.

[10] S. Har-Peled. A replacement for Voronoi diagrams of near linear size. In *Proc. 42nd IEEE Sympos. Found. Comput. Sci.*, pages 94–103, 2001.

[11] P. Indyk. Nearest neighbors in high-dimensional spaces. In *Handbook of Discrete and Computational Geometry* (J. E. Goodman and J. O'Rourke, ed.), CRC Press, 2nd ed., 2004.

[12] S. Liao, M. A. Lopez, and S. T. Leutenegger. High dimensional similarity search with space filling curves. In *Proc. of Int. Conf. on Data Engineering*, 2001. `http://www.cs.du.edu/~leut/icde01.ps`.

[13] M. A. Lopez and S. Liao. Finding k-closest-pairs efficiently for high dimensional data. In *Proc. 12th Canad. Conf. Comput. Geom.*, pages 197–204, 2000.

[14] D. M. Mount and S. Arya. ANN: a library for approximate nearest neighbor searching. `http://www.cs.umd.edu/~mount/ANN/`, version 1.1, 2005.

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

[16] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

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

[18] M. Smid. Closest-point problems in computational geometry. In *Handbook of Computational Geometry* (J. Urrutia and J. Sack, ed.), North-Holland, pages 877–935, 2000.

Appendix: Code

```
// Timothy Chan  12/05
// approximate nearest neighbors: the SSS method (static version)

#include <stream.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>
#define sq(x) (((float) (x))*((float) (x)))
#define MAX (1<<29)

typedef int *Point;
int d, shift;
float eps, r, r_sq;
Point ans, q1, q2;

inline int less_msb(int x, int y) { return x < y && x < (x^y); }

int cmp_shuffle(Point *p, Point *q) {
  int j, k, x, y;
  for (j = k = x = 0; k < d; k++)
    if (less_msb(x, y = ((*p)[k]+shift)^((*q)[k]+shift))) {
      j = k; x = y;
    }
  return (*p)[j]-(*q)[j];
}

void SSS_preprocess(Point P[], int n) {
  shift = (int) (drand48()*MAX);
  q1 = new int[d]; q2 = new int[d];
  qsort((void *) P, n, sizeof(Point),
      (int (*)(const void *, const void *)) cmp_shuffle);
}

void check_dist(Point p, Point q) {
  int j; float z;
  for (j = 0, z = 0; j < d; j++) z += sq(p[j]-q[j]);
  if (z < r_sq) {
    r_sq = z; r = sqrt(z); ans = p;
    for (j = 0; j < d; j++) {
      q1[j] = (q[j]>r) ? (q[j]-(int)ceil(r)) : 0;
      q2[j] = (q[j]+r<MAX) ? (q[j]+(int)ceil(r)) : MAX;
    }
  }
}

float dist_sq_to_box(Point q, Point p1, Point p2) {
  int i, j, x, y; float z;
  for (j = x = 0; j < d; j++)
    if (less_msb(x, y = (p1[j]+shift)^(p2[j]+shift))) x = y;
  frexp(x, &i);
  for (j = 0, z = 0; j < d; j++) {
    x = ((p1[j]+shift)>>i)<<i; y = x+(1<<i);
    if (q[j]+shift < x) z += sq(q[j]+shift-x);
    else if (q[j]+shift > y) z += sq(q[j]+shift-y);
  }
  return z;
}

void SSS_query0(Point P[], int n, Point q) {
  if (n == 0) return;
```

```
  check_dist(P[n/2], q);
  if (n == 1 || dist_sq_to_box(q, P[0],P[n-1])*sq(1+eps) > r_sq) return;
  if (cmp_shuffle(&q, &P[n/2]) < 0) {
    SSS_query0(P, n/2, q);
    if (cmp_shuffle(&q2, &P[n/2]) > 0) SSS_query0(P+n/2+1, n-n/2-1, q);
  }
  else {
    SSS_query0(P+n/2+1, n-n/2-1, q);
    if (cmp_shuffle(&q1, &P[n/2]) < 0) SSS_query0(P, n/2, q);
  }
}

Point SSS_query(Point P[], int n, Point q) {
  r_sq = FLT_MAX;
  SSS_query0(P, n, q);
  return ans;
}

main(int argc, char *argv[]) {
  int n, m, i, j;
  Point *P, q;
  eps = (argc == 2) ? atof(argv[1]) : 0;
  cin >> n; cin >> m; cin >> d;
  srand48(12121+n+m+d);
  P = new Point[n]; q = new int[d];
  for (i = 0; i < n; i++) {
    P[i] = new int[d];
    for (j = 0; j < d; j++) cin >> P[i][j];
  }
  SSS_preprocess(P, n);
  for (i = 0; i < m; i++) {
    for (j = 0; j < d; j++) cin >> q[j];
    SSS_query(P, n, q);
    cout << r << "\n";
  }
  for (i = 0; i < n; i++) delete P[i];
  delete P; delete q;
}
```