Fast nearest-neighbor searching for nonlinear signal processing

Christian Merkwirth,* Ulrich Parlitz, and Werner Lauterborn
Drittes Physikalisches Institut, Universität Göttingen, Bürgerstraße 42-44, D-37073 Göttingen, Germany
(Received 15 October 1999; revised manuscript received 29 March 2000)

A fast algorithm for exact and approximate nearest-neighbor searching is presented that is suitable for tasks encountered in nonlinear signal processing. Empirical benchmarks show that the algorithm's performance depends mainly on the (fractal) dimension D_d of the data set, which is usually smaller than the dimension D_s of the vector space in which the data points are embedded. We also compare the running time of our algorithm with those of two previously proposed algorithms for nearest-neighbor searching.

PACS number(s): 05.45.−a, 07.05.Kf

I. INTRODUCTION

The task of finding one or more nearest neighbors in a D_s-dimensional space occurs in many fields of data processing, e.g., information retrieval in database applications, data mining, or, as in our case, nonlinear time-series analysis [1–3], especially for modeling and prediction of time series (via time-delay reconstruction) [4], fast correlation sum computation (correlation dimension, generalized mutual information, etc.), estimation of the Renyi dimension spectrum [5] or Lyapunov exponents [4], and nonlinear noise reduction [6].

Nearest-neighbor searching and related problems of computational geometry have been extensively studied in computer science and pattern recognition and turned out not to fall into the class of computationally hard problems. Searching the nearest neighbor to every point in a data set X of size N using a naive algorithm (which calculates all interpoint distances) is of order $O(N^2)$. However, for practical applications it would be useful to have an algorithm of order $O(N \log N)$. While the specific details of the algorithms that have been proposed so far vary, a common approach is to build up an auxiliary indexing data structure in the *preprocessing phase* which helps finding nearest neighbors during the *search phase*.

II. ATRIA: A TRIANGLE INEQUALITY BASED ALGORITHM

The algorithm to be presented is based on the ANNA algorithm proposed by McNames [7]. To prune the search space, the algorithm employs the triangle inequality:

$$d(x,z) \leqslant d(x,y) + d(y,z)$$

for any triple of points x, y, and z.

Here $d(x,y)$ denotes the distance between points x and y.

Since this inequality is valid for any metric, there is no limitation in what kind of metric is used to calculate distances. We successfully used the algorithm to compute neighbors in spherical geometry. Additionally, the preprocessing is independent of the type of queries (exact or approximate) that will be executed during the search phase.

Another advantage of the proposed algorithm is its relatively low complexity. For data sets of fixed fractal dimension D_d, search time grows approximately linearly with the dimension D_s of the data space [7]. The typical (not worst case) memory consumption is of order $O(N)$.

III. PREPROCESSING PHASE

A. Top-down construction of the cluster tree

During the preprocessing phase, a hierarchical cluster tree (Fig. 1) is recursively constructed. A *cluster* contains a distinct subset of the points of the original data set. Additionally, it is characterized by its center point c and the minimal radius R that is needed to cover all points belonging to this cluster. A new level of the binary tree is constructed by dividing (splitting) each cluster of the current level into two child clusters (*subclusters*). At any tree level, each point of the data set is an element of exactly one cluster.

B. Clustering strategy

In the following section we present the main steps of the recursive clustering algorithm.

(1) Choose two cluster centers, one for each of the two new child clusters. For this choice, many different selection schemes are possible. We propose a very simple one: If the current cluster is the root cluster, pick randomly a point a out

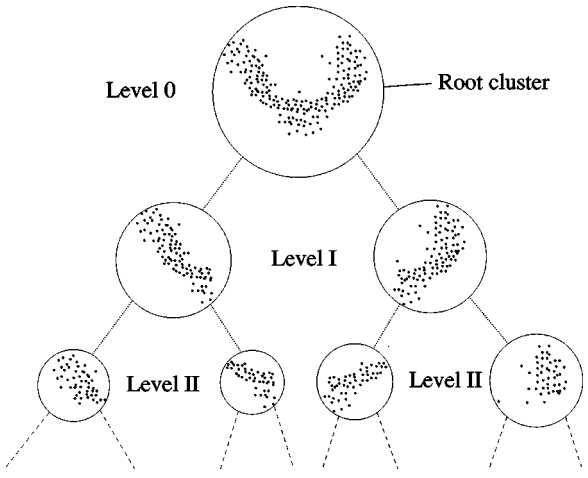

FIG. 1. First levels of the hierarchical cluster tree.

*Email address: C.Merkwirth@dpi.physik.uni-goettingen.de

of the points belonging to this cluster. Otherwise, let a be the current cluster's center. Then determine the point c_r with maximal distance from a. This point becomes the center of the first (right) child cluster. Now the center c_l of the second (left) child cluster is the point that has maximal distance to c_r.

(2) Assign each point of the current cluster to one of its child clusters. As criterion the distance to the child cluster centers is used: Choose the nearer one.

(3) Compute and store the enclosing radii R of both newly created subclusters.

(4) Proceed recursively as long as a cluster contains more than L points ($L \approx 30, \ldots, 200$). A cluster that is not further divided into subclusters is called a *terminal node* of the cluster tree.

(5) When a terminal node is reached, compute and store the distances from the cluster's center to all points belonging to this cluster.

C. Implementation details

Points of the input data set are addressed using an integer index, ranging from 1 to N. To handle these indices, it is not necessary to store an array of size N for each level of the cluster tree, actually, we just need to hold one linear array A of size N in memory. The indices of the points belonging to the same cluster are stored in a contiguous section of the array A.

The data structure that represents one cluster stores the following attributes

c is the index of the point that is designated as this cluster's center.

R the maximal distance from the center point to any point belonging to this cluster,

$$R = \max_{x \in C} d(c,x).$$

g the points belonging to the parent cluster are assigned to this cluster when their distance to this cluster's center is smaller than their distance to the center of this cluster's sister. During preprocessing, the minimum of the difference of the distances is computed and stored in g:

$$g = \min_{x \in C}[d(c_{sister(i)},x) - d(c,x)].$$

start is the beginning of the section on A where the indices of this cluster's points are stored.

end is the end of the section on A where the indices of this cluster's points are stored.

left, **right** are pointers to the left and right child clusters (used only in internal nodes).

The root cluster's **start** value points to the beginning of A, its **end** value to A's end. To split a cluster into subclusters, it is not necessary to alter the start and end of the section where the indices of the current cluster's points are stored. Only permutations of the indices inside this section are needed (see Fig. 2). These permutations can be performed using a quicksort-like scheme [8] to assign points to the nearest cluster center (Fig. 3).

Storing distances to the center for points in terminal nodes can also be done using just one linear array of length N. All section boundaries are the same as for the index array A. Our implementation of the algorithm uses an array of tupels (index, distance) to efficiently access both quantities.

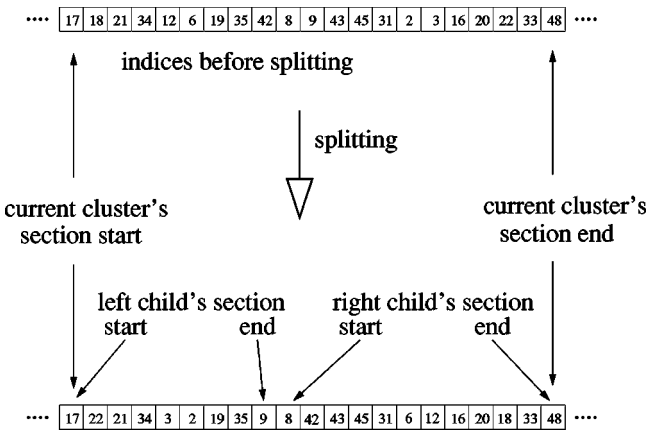

FIG. 2. Splitting the current cluster. The section boundaries of the current cluster do not change during the splitting. Entries are exchanged only within the current section to segregate the indices of the left and right child clusters.

IV. SEARCH PHASE

A. Prune and conquer

Searching for one or more nearest neighbors can be trivially done by computing distances from the query point q to all points of the data set and sorting these distances in increasing order. However, this leads to a time complexity of $O(N)$ for a single query. In order to reduce the number of distance calculations, we have to prune the search tree. This can be done by excluding clusters from the search that are very far away from the query point and cannot therefore contain a nearest neighbor. To quantify this rule, the algorithm has to maintain a sorted table of points m_i ($i = 1, \ldots, k$) that are close to q. These *preliminary neighbors* will be successively replaced by better (nearer) candidates until the final set of nearest neighbors is found.

Once a few preliminary neighbors have been inserted into this table, the value $d(m_k,q)$ (distance from query point q to the kth nearest point found so far) is given, which is an upper limit of the distance to the actual kth nearest neighbor. $d(m_k,q)$ is used within three rules.

(1) Exclude cluster i if

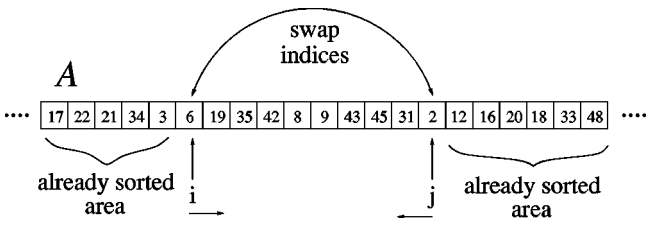

FIG. 3. Exchanging indices inside current section. Indices of points that are (geometrically) closer to the left (right) child's center are moved to the left (right) side of the section. During sorting, pointers i and j move inward from the current cluster's section boundaries, stopping at indices that need to be exchanged. The position where both pointers meet defines the section boundaries of the new child clusters.

$$d(m_k,q)<\hat{d}_{min}(i) \quad (1)$$

where $\hat{d}_{min}(i)$ gives a lower bound on the distances from the query point to any point inside cluster i (for details, see Sec. IV B).

(2) If cluster i is a terminal node, exclude any point x belonging to this cluster from being searched if

$$d(m_k,q)<|d(c_i,q)-d(c_i,x)|.$$

Values $d(c_i,x)$ were computed and stored during the preprocessing phase (for details, see Sec. III C and Appendix, proof 3).

(3) Partial distance calculation: To qualify a point x as nearest-neighbor candidate, its distance $d(x,q)$ must be smaller than the actual threshold $d(m_k,q)$, otherwise the exact value of $d(x,q)$ is not needed. This allows one to terminate the operation of calculating $d(x,q)$ as soon as the partial distance exceeds $d(m_k,q)$ (details depend on the kind of distance function used) [9]. Carefully implemented, partial distance calculation can seamlessly be integrated into the search phase to further reduce computational effort. As in the previous case (2), this rule is applied only when scanning through the points inside a terminal node.

B. Computing $\hat{d}_{min}$

To determine whether cluster i needs to be searched, we need an estimate $\hat{d}_{min}(i)$ of the (unknown) actual smallest distance $d_{min}(i)=\min_{x\in C_i} d(x,q)$ from the query point to any point inside this cluster so that $0\leqslant\hat{d}_{min}(i)\leqslant d_{min}(i)$. Three lower bounds on d_{min} can easily be computed, using a mixture of information from the preprocessing phase and distances calculated while processing a query.

(1) $d(c_i,q)-R_i$. Here $d(c_i,q)$ denotes the distance between the query point q and the center c of cluster i, while R_i is the cluster's radius (see Appendix, proof 1).

(2) $\frac{1}{2}[d(c_i,q)-d(q,c_{sister(i)})+g_i]$, where $d(q,c_{sister(i)})$ denotes the distance from the query point to the center of cluster i's sister (see Appendix, proof 2).

(3) Using the nesting property of the hierarchical clustering, a cluster's d_{min} cannot be smaller than its parent's $\hat{d}_{min}$.

Since all three values are valid lower bounds on the actual minimum distance, their maximum is assigned to $\hat{d}_{min}(i)$.

It can hardly be overstressed that estimating d_{min} is a very crucial part of this algorithm. If we had an exact estimate, we could selectively pick out only those terminal nodes that actually contain one or more nearest neighbors (these are maximal k clusters). Unfortunately, it seems that it is not possible to compute the exact value of d_{min} without explicitly calculating distances $d(x,q)\forall x$ of the given cluster C. Then, however, it is too late to use d_{min} to prune the search tree since the work that should be avoided is already done.

We formulated all inequalities that arose during the traversal of the cluster tree into a linear programming problem, which, to our surprise, did not yield better estimates for d_{min} than those that were obtained from the rules 1–3 described above.

C. Priority queue controlled search order

The order in which the nodes of the tree are searched is crucial to minimize the overall number of distance calculations. Usually, a stack is used to control the tree traversal (depth first). The disadvantage of this method is that, once a cluster is inserted into the stack, it is not possible to alter the order in which the clusters are processed, even if a more promising path through the cluster tree is discovered in the meantime.

To ensure that clusters are visited in order of increasing distance to q, the proposed algorithm employs a *priority queue* (PQ) [8,10]. The PQ guides the traversal through the tree by keeping track of the nodes that have not been searched yet. The PQ's entries are sorted by $\hat{d}_{min}$ so that the cluster with the smallest $\hat{d}_{min}$ can efficiently be extracted from the top of the PQ. Due to rule 2 in Sec. IV B, there is (if any) one path from the root cluster to one of the terminal nodes with $\hat{d}_{min}=0$ (the node the query point would fall into if it were part of the preprocessing); for all other terminal nodes $\hat{d}_{min}$ is strictly positive [except for rather degenerate queries where $d(c_i,q)$ equals $d(c_{sister(i)},q)$ for all pairs of child clusters].

D. Processing a query

This section outlines the search algorithm's main loop and the next two give the termination criteria for exact and approximate queries. The main loop proceeds as follows.

Initialize the sorted table of preliminary neighbors $(m_1,\ldots,m_k)$.

Insert the root cluster into the priority queue.

Repeat the following sequence.

(i) Extract cluster C with the smallest $\hat{d}_{min}$ from the PQ.

(ii) Break if the termination criterion is reached.

(iii) If C is a terminal node, scan through the cluster by computing $d(x,q)\ \forall x\in C$, obeying rule 2 (see Sec. IV A), and insert x into a sorted table of preliminary neighbors if $d(x,q)\leqslant d(m_k,q)$.

(iv) Otherwise insert child clusters of C into the PQ.

E. Exact queries

Once all clusters have been visited with

$$\hat{d}_{min}\leqslant d(m_k,q),$$

no *better* (i.e., nearer) neighbor can be found in one of the remaining clusters. This is illustrated in Fig. 4, where both the decreasing distance to the kth preliminary neighbor and the estimate $\hat{d}_{min}$ of the minimal distance between the q and the cluster's points are plotted. When both curves intersect, the algorithm can terminate and return the preliminary neighbors $m_1,\ldots,m_k$ as exact nearest neighbors $n_1,\ldots,n_k$.

F. Approximate queries

Approximate queries can be seen as an extension to the exact problem. If one is not interested in getting the *k nearest* neighbors, but just k neighbors that are not too far off the exact ones (in terms of their distances), approximate queries can greatly reduce the search time. An additional input pa-

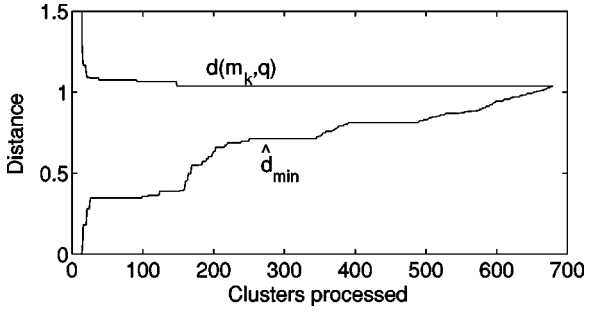

FIG. 4. Typical progression of $d(m_k,q)$ and $\hat{d}_{min}$ for an exact query. The cluster tree contains more than 2300 nodes. The termination criterion is reached when both curves intersect. The long flat part of the $d(m_k,q)$ curve shows that no better neighbors are found after the first 147 clusters have been processed although the $\hat{d}_{min}$ curve is still lower until cluster 679 is searched.

rameter ϵ has to be provided that specifies the maximal allowed error in the returned neighbor distances $d(n_i^{\epsilon},q)$ with respect to the distance of the true nearest neighbors $d(n_i^0,q)$ [11]:

$$d(n_i^{\epsilon},q) \leqslant (1+\epsilon)d(n_i^0,q), \quad i=1,\ldots,k. \tag{2}$$

For $\epsilon \to 0$, an approximate query becomes an exact query. Approximate queries return before all clusters that might contain exact nearest neighbors have been visited, which speeds up the calculation, but might result in wrongly reported nearest neighbors. However, often many returned approximate neighbors are in fact exact ones and the average relative error in the distances is one or two orders smaller than ϵ would allow (see Sec. V B, dashed line in Fig. 7 below).

Approximate queries can be implemented within the framework of the previously outlined search algorithm by replacing the termination criterion with

$$\hat{d}_{min} > \frac{d(m_k,q)}{1+\epsilon} \tag{3}$$

(see Appendix, proof 4).

V. EXPERIMENTAL RESULTS

All benchmark and timing measurements were performed on a Silicon Graphics Indigo2, running IRIX 6.5 as operating system. The system was equipped with a 175 MHz Mips R1000 CPU, 256 Mbytes main memory and 1 Mbyte secondary level cache. Background load was kept as constant and minimal as possible on a multiuser system to ensure unbiased test conditions. Distances were, if not otherwise stated, computed using the L_2 metric. The only user-specified parameter L of the ATRIA algorithm was set at 64 for all tests, which served as a good trade-off between tree depth and cluster localization in past experiments. Empirically, little advantage could be drawn from optimizing this parameter for each data set individually.

The data sets used in this section were generated by three different dynamical systems.

Data sets of type A were generated by a D_s-dimensional generalization of the iterated Hénon map [12]:

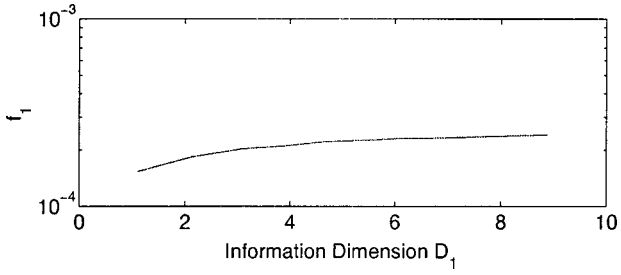

FIG. 5. Relative number of distance calculations needed to locate a point versus information dimension D_1. The graph gives f_1 for data sets of type A with D_s varying from 2 to 12. For this system, the information dimension D_1 grows monotonically with D_s. Each data set consisted of 200 000 points. The query points were randomly chosen from the data set itself; self-matches were explicitly allowed.

$$(x_1)_{n+1} = a - (x_{D_s-1})_n^2 - b(x_{D_s})_n,$$

$$(x_i)_{n+1} = (x_{i-1})_n, \quad i=2,\ldots,D_s,$$

with $a=1.76$ and $b=0.1$. Starting from random initial conditions, we discarded the first 5000 iterations and used the next 200 000 iterations.

Data sets of type B were generated using a hyperchaotic generalization of the Rössler system that was introduced by Baier and Sahle [13]:

$$\dot{x}_1 = -x_2 + a x_1,$$

$$\dot{x}_i = x_{i-1} - x_{i+1}, \quad i=2,\ldots,M-1,$$

$$\dot{x}_M = \epsilon + b x_M (x_{M-1} - d),$$

with parameters $a=0.28$, $b=4$, $d=2$, and $\epsilon=0.1$. The system was integrated from $T=0$ to $T=21\,200$ and the system

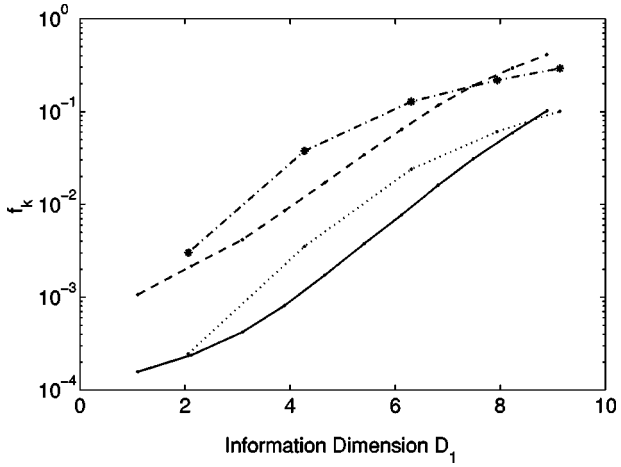

FIG. 6. Relative number of distance calculations versus information dimension D_1. The solid curve denotes f_1 for data sets of type A with D_s varying from 2 to 12. For this system, the information dimension D_1 grows steadily with D_s. The dashed line shows f_{128} for the same data sets. The dotted curve depicts f_1 for data sets of type B with fixed embedding dimension $D_s=24$, but M varying from 3 to 11 (odd values only). The dashed-dotted line shows f_{128} for the same data sets. Each data set consisted of 200 000 points.

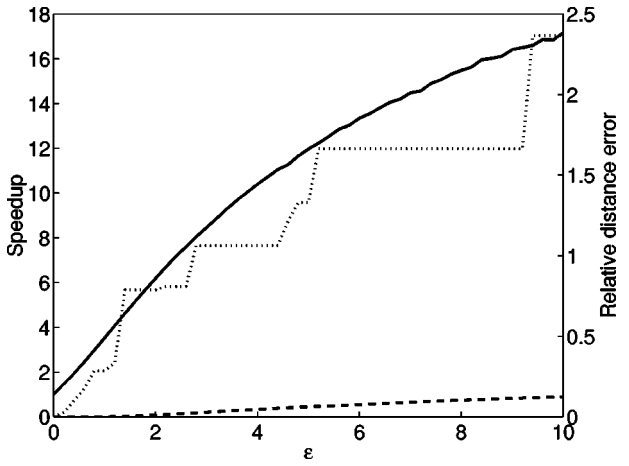

FIG. 7. Speedup in search time and relative distance error versus approximation parameter ϵ for 50 000 data points of type A with $D_s=8$ and $D_1\approx 5.9$. The solid line denotes the speedup factor. The dashed line shows the averaged relative distance error. The maximal relative error that was encountered is given by the dotted line.

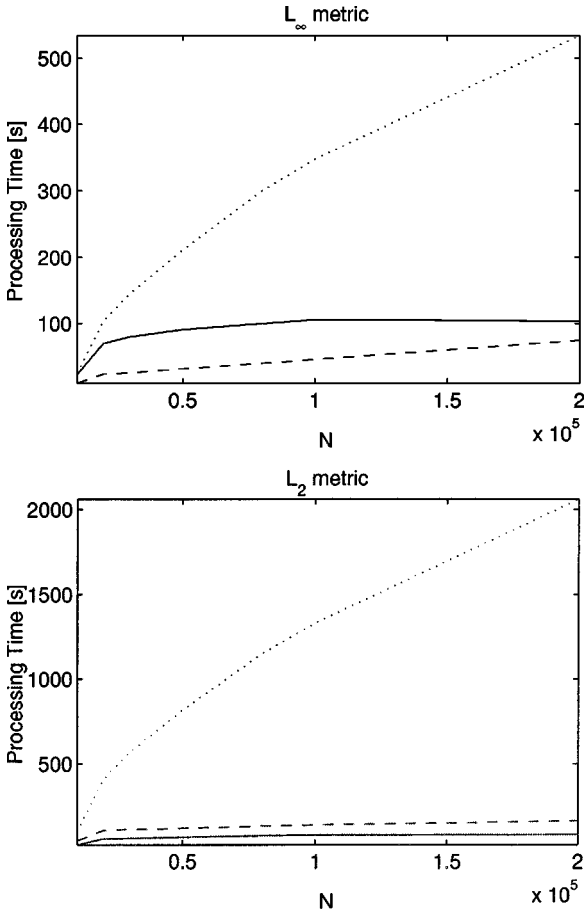

FIG. 8. Sum of preprocessing and search time for box-assisted nearest-neighbor search (dotted curve), *kd* tree (dashed curve), and the proposed ATRIA algorithm (solid curve) versus size N of data set B with $M=5$. Twelve exact nearest neighbors in the L_∞ and L_2 metrics had to be found to 20 000 reference points which were chosen from the data set points, except for the smallest data set size of 10 000 points, where each point was used as a query point. Self-matches were excluded. For detailed timing values, see Table I.

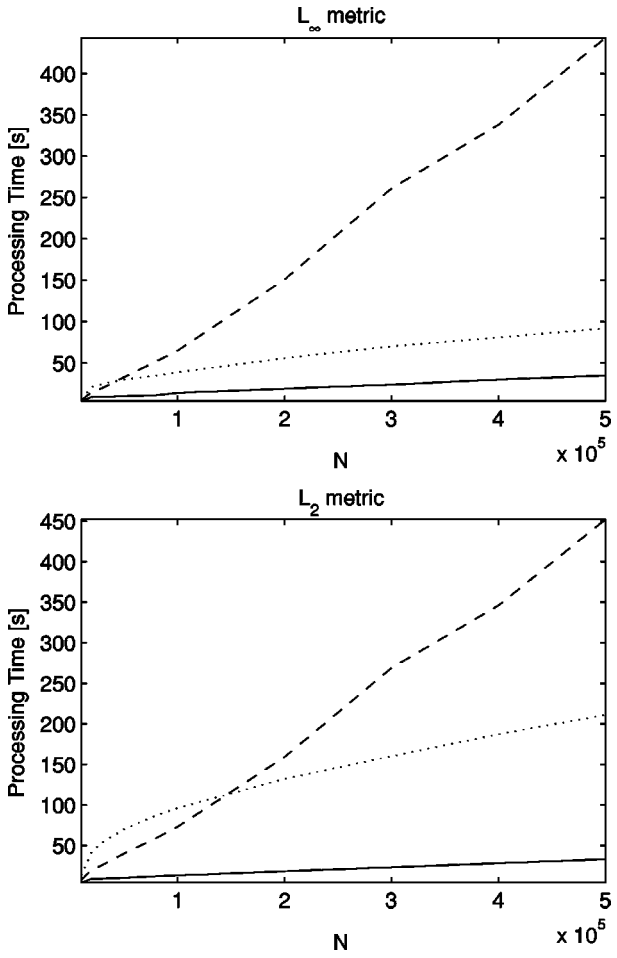

FIG. 9. Sum of preprocessing and search time for box-assisted nearest-neighbor search (dotted curve), *kd* tree (dashed curve), and the proposed ATRIA algorithm (solid curve) versus size N of data set C. Twelve exact nearest neighbors in the L_∞ and L_2 metrics had to be found to 20 000 reference points which were chosen from the data set points, except for the smallest data set size of 10 000 points, where each point was used as a query point. Self-matches were excluded. For detailed timing values, see Table II.

state was sampled with $\triangle T=0.1$. The first 10 000 transient samples were discarded. Then x_1 was used for a time-delay reconstruction with an embedding dimension of $D_s=24$ and a time delay of $9\triangle T$. The resulting data set was cropped to 200 000 time-delay vectors.

Data set C was generated using the Lorenz system of three ordinary differential equations:

$$\dot{x}_1=\sigma(x_1-x_2),$$

$$\dot{x}_2=rx_1-x_2-x_1x_3,$$

$$\dot{x}_3=x_1x_2-bx_3,$$

with parameters $\sigma=-10$, $b=-\frac{8}{3}$, and $r=28$. The system was integrated from $T=0$ to $T=1500$ and the system state was sampled with $\triangle T=0.025$. The first 40 000 transient samples were discarded. Then x_1 was used for a time-delay reconstruction with an embedding dimension of $D_s=25$ and a time delay of $\triangle T$. The resulting data set was cropped to 500 000 time-delay vectors.

TABLE I. Comparison of *kd* tree (ANN), box-assisted nearest-neighbor search (BOX), and ATRIA on data set *B* with $M=5$ and $D_1 \approx 4.3$. N denotes the number of data set points. Twelve exact nearest neighbors in both the L_∞ and L_2 metrics had to be found to N_{ref} query points which were chosen from the data set; therefore self-matches were excluded. For each metric two values are given. The left one specifies the preprocessing time, the right gives the total search time. Timing values are measured in seconds with a resolution of one second, so a value of zero seconds just indicates a time smaller than one second.

		t_{ANN}				t_{BOX}				t_{ATRIA}			
		L_∞		L_2		L_∞		L_2		L_∞		L_2	
$\frac{N}{10^4}$	$\frac{N_{\mathrm{ref}}}{10^4}$												
1	1	1	9	0	44	0	25	0	99	1	22	0	20
2	2	3	21	2	105	0	104	0	408	1	69	0	56
3	2	4	22	4	108	0	145	0	568	1	79	1	58
5	2	8	24	8	111	0	211	0	817	2	89	1	65
8	2	15	25	14	118	0	300	0	1149	3	97	3	72
10	2	19	27	20	118	1	347	1	1330	5	101	4	74
20	2	47	28	47	117	1	533	1	2060	9	95	9	73

A. Computational complexity versus the data set's fractal dimension

To give a hardware independent index for the computational costs of nearest-neighbor searching, we use the fraction f_k of distance calculations needed to find k neighbors of a given query point, divided by the total number of points and averaged over a large number of queries. Since the query points were randomly chosen from each data set, self-matches had to be excluded (locating an existing point can be performed almost regardless of both the data space dimension and the fractal dimension of the data set; see Fig. 5).

For the brute force method f_k equals 1 (the distances to all points of the data set have to be computed in order to find the nearest neighbors). An efficient algorithm should yield fractions f_k much smaller than 1.

Figure 6 shows f_1 and f_{128} for data sets of types *A* and *B*. While the data space dimension D_s varies from 2 to 12 for the data sets of type *A*, it is held constant at 24 for the data sets of type *B*, where the system dimension M is increased from 3 to 11 (odd values only) to produce data sets of varying fractal dimension D_d. As a measure for D_d, we employ the *information* dimension D_1 [14].

The tests consisted in searching the exact nearest neighbor or the 128 exact nearest neighbors of each of the 5000 query points. Note the logarithmic scale of the f_k axis. The number of distance calculations grows almost exponentially with D_1. At higher D_1, the increase starts to slow down because the number of distance calculations needed to find the nearest neighbors approaches the total number of data set points. Then, of course, f_k converges to 1 and there is no advantage in preferring this algorithm over a brute force approach.

Despite the very different dimension D_s of the data space, the resulting curves f_1 and f_{128} are quite similar for both types of data sets. Focusing on the first intersection of the f_1 curves, data set *A*'s data space dimension D_s equals 3, while data set *B*'s D_s is 24. This clearly indicates that this dimension is of little importance for the search efficiency of the algorithm. Of course, the absolute search time grows with

TABLE II. Comparison of *kd* tree (ANN), box-assisted nearest-neighbor search (BOX), and ATRIA on data set *C* with $D_1 \approx 2.05$. N denotes the number of data set points. Twelve exact nearest neighbors in both the L_∞ and L_2 metrics had to be found to N_{ref} query points which were chosen from the data set; therefore self-matches were excluded. For each metric two values are given. The left one specifies the preprocessing time, the right gives the total search time. Timing values are measured in seconds with a resolution of one second, so a value of zero seconds just indicates a time smaller than one second.

		t_{ANN}				t_{BOX}				t_{ATRIA}			
		L_∞		L_2		L_∞		L_2		L_∞		L_2	
$\frac{N}{10^4}$	$\frac{N_{\mathrm{ref}}}{10^4}$												
1	1	2	3	1	6	0	7	0	13	1	3	0	4
2	2	5	10	5	15	0	21	0	42	1	8	1	8
3	2	11	9	10	16	1	24	0	54	1	8	1	8
5	2	22	11	23	17	1	29	0	70	2	8	2	8
8	2	41	11	41	18	0	35	1	86	3	8	3	9
10	2	54	11	55	18	0	39	0	96	5	9	4	9
20	2	140	11	140	19	1	55	1	131	9	10	8	10
30	2	249	12	249	20	1	69	1	159	14	10	13	10
40	2	326	12	326	20	2	79	2	185	20	10	18	10
50	2	430	13	428	24	2	90	2	209	25	10	23	10

increasing D_s even if f_k remains constant because of the increasing number of arithmetic operations needed for each distance calculation.

The particular choice of D_1 out of the continuous spectrum of Renyi dimensions D_q is arbitrary and was motivated mainly by the fact that D_1 can easily be computed from the distribution of neighbor distances [5]. However, D_1 does not at all describe the distribution of the data set points completely and may thus only serve as an estimator for the expected perfomance of the nearest-neighbor algorithm.

B. Speedup and relative error for increasing the approximation parameter ϵ

Fifty thousand points of data set A with $D_s=8$ and $D_d \approx 5.9$ were used to calculate speedup and errors introduced by approximate queries. The task consisted in finding the eight nearest neighbors (excluding self-matches) to 10 000 query points that were chosen randomly out of the data set.

In Fig. 7, the solid line depicts the speedup obtained using approximate queries instead of exact queries for increasing values of ϵ (in the case of $\epsilon=0$, approximate queries are exact ones). The speedup factor was calculated by dividing the execution time of the exact queries by the execution time of the corresponding approximate queries (preprocessing was done only once prior to all timing measurements). The dashed line shows the relative distance error, averaged over all neighbors and queries. The dotted line denotes the maximum relative error of the reported nearest-neighbor distances. For ϵ as large as 7, the average relative error in the neighbor distances does not exceed 10%. This is astoninglishly small since $\epsilon=7$ would allow an error up to 700%. Again, this discrepancy seems to be a consequence of the inability to estimate d_{min} exactly.

C. Empirical benchmarks

In this section, we compare the ATRIA algorithm with two implementations of previously proposed nearest-neighbor algorithms, the *kd* tree and box-assisted nearest-neighbor search [3], on data sets of types *B* and *C*. Distances were calculated in the L_∞ and L_2 metrics.

The implementation of the *kd* tree with a sliding midpoint splitting method was taken from the beta release of ANN (version 0.2) by the University of Maryland [15]. The box-assisted nearest-neighbor algorithm was implemented by the authors. All code was compiled using the same compiler options and optimization settings. The output of the three algorithms was checked, but except for permutations of the indices of neighbors within the same distance to q no differences were found.

Benchmarks (see Figs. 8 and 9 and Tables I and II) were performed for data sets of increasing size. For the smallest data set size, 10 000 query points were used, for all other sizes 20 000 reference (=query) points were chosen from the data set. The timing values include preprocessing and search time for exact nearest-neighbor queries with $k=12$.

Though the ATRIA algorithm is clearly the fastest on data set *C*, it is outperformed by the *kd* tree algorithm on data set *B* when distances are calculated in the L_∞ metric (see Fig. 8). Especially for small data set sizes, the ATRIA algorithm needs a significantly longer time to execute the neighbor queries. However, one must keep in mind that ATRIA is not optimized for one special kind of metric, while the coordinatewise splitting employed in the *kd* tree combines well with the computation of distances in the L_∞ metric. In fact, the *kd* tree's bad performance on data sets of type *C* is caused mainly by the very long preprocessing time required (see Table II), which may indicate an unlucky choice of the splitting rule (the splitting rule recommended by the authors of the ANN package was used). But even with one second rank the ATRIA algorithm still delivers comparable and stable timing results with a moderate amount of preprocessing, which can be important when the number of queries is much smaller than the number of data set points.

VI. DISCUSSION

We presented an efficient nearest-neighbor search algorithm that performs well for data sets with small fractal dimension, which are frequently encountered in the field of nonlinear signal processing. Another advantage of this algorithm is that no explicit coordinate representation of the data set or query points is required. Instead, a method for calculating distances must be provided, which can rely on an arbitrary metric. This allows use of this algorithm for nearest-neighbor problems in rather unusual spaces, e.g., function spaces or spaces of data base objects, even kernel functions similar to the ones used in support vector machines [16] could be used to efficiently compute distances in extremely high-dimensional feature spaces. Approximate neighbor queries optionally offer the possibility of speeding up the search operation in tradeoff for small errors in the returned neighbor distances.

When neighbors are searched in data sets of high fractal dimension, all interpoint distances become very similar (*concentration of measure* [17]) and an effective preselection of clusters is no longer possible. Due to the additional overhead of preprocessing and managing search order, the total computation time can then be even larger than that of a brute force approach.

The presented algorithm for nearest-neighbor searching is available as part of the software package TSTOOL at http://www.physik3.gwdg.de/tstool

ACKNOWLEDGMENT

The authors acknowledge financial support by the Bundesministerium für Bildung und Forschung, Grant No. 13N7038/9.

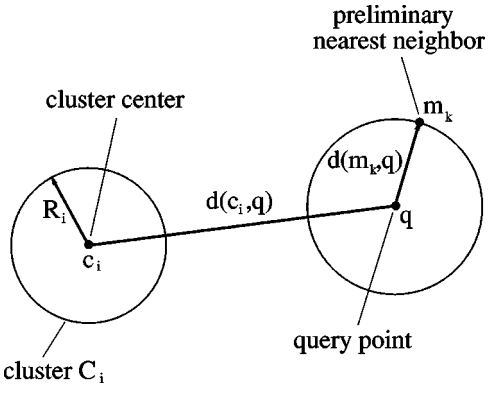

FIG. 10. Illustration for proof 1.

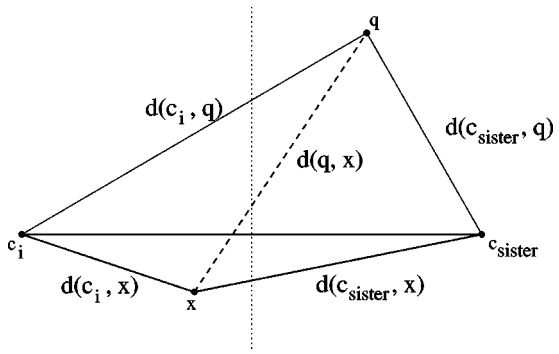

FIG. 11. Illustration for proof 2.

APPENDIX

Proof 1. By construction, we know that $\forall x$ belonging to the current cluster C_i (see Fig. 10)

$$d(c_i,x) \leqslant R_i .$$

From the triangle inequality it follows that

$$d(q,c_i) \leqslant d(c_i,x) + d(q,x),$$

$$d(q,c_i) \leqslant R_i + d(q,x) \Rightarrow d(q,c_i) - R_i \leqslant d(q,x).$$

Proof 2. By construction, we know that $\forall x$ belonging to the current cluster C_i (see Fig. 11)

$$d(c_i,x) + g_i \leqslant d(c_{sister(i)},x). \tag{A1}$$

From the triangle inequality it follows that

$$d(c_{sister(i)},x) \leqslant d(q,c_{sister(i)}) + d(q,x), \tag{A2}$$

$$d(c_i,q) \leqslant d(c_i,x) + d(q,x), \tag{A3}$$

$$(\text{A1}) + (\text{A2}) \Rightarrow d(c_i,x) + g_i \leqslant d(q,c_{sister(i)}) + d(q,x), \tag{A4}$$

$$(\text{A3}) \Rightarrow d(c_i,q) - d(q,x) \leqslant d(c_i,x), \tag{A5}$$

$$(\text{A4}) + (\text{A5}) \Rightarrow d(c_i,q) - d(q,c_{sister(i)}) + g_i \leqslant 2d(q,x).$$

Proof 3. From the triangle inequality we know that $\forall x$ belonging to the current cluster C_i (see Fig. 12)

$$d(c_i,x) \leqslant d(c_i,q) + d(q,x),$$

$$d(c_i,q) \leqslant d(c_i,x) + d(q,x),$$

$$\Rightarrow |d(c_i,q) - d(c_i,x)| \leqslant d(q,x).$$

Proof 4. Let $n_i^0 \in X$ $(i=1,\ldots,k)$ be the exact nearest neighbors to query point q such that

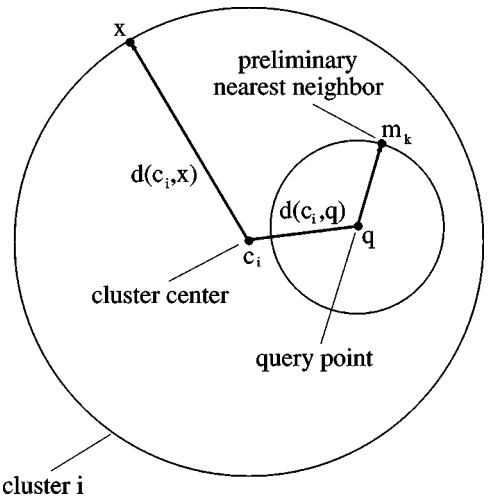

FIG. 12. Illustration for proof 3.

$$0 \leqslant d(n_i^0,q) \leqslant d(n_{i+1}^0,q), \quad i=1,\ldots,k-1,$$

$$d(n_k^0,q) \leqslant d(x,q) \forall x \in X \backslash \{n_1^0,\ldots,n_k^0\}.$$

Thus, we know for any sorted set of preliminary neighbors that

$$d(n_i^0,q) \leqslant d(m_i,q), \quad i=1,\ldots,k.$$

If the search terminates when the criterion

$$\hat{d}_{min} > \frac{d(m_k,q)}{1+\epsilon}$$

is reached, all preliminary neighbors with $d(m_i,q) \leqslant d(m_k,q)/(1+\epsilon)$ are exact ones and there are no more exact neighbors closer to q than this threshold, since all points belonging to clusters up to this distance have been exhaustively searched. Let j denote the highest index i for which this condition holds, otherwise set j to 0. For $i=1,\ldots,j$, $m_i = n_i^0$ and therefore condition (2) of Sec. IV F is trivially fulfilled. We also know that

$$\frac{d(m_k,q)}{1+\epsilon} \leqslant d(m_{j+1},q) \leqslant \cdots \leqslant d(m_k,q) \tag{A6}$$

as well as

$$\frac{d(m_k,q)}{1+\epsilon} \leqslant d(n_{j+1}^0,q) \leqslant \cdots \leqslant d(n_k^0,q)$$

$$\leqslant d(m_k,q) \Rightarrow d(m_k,q) \leqslant (1+\epsilon) d(n_{j+1}^0,q). \tag{A7}$$

Combining Eqs. (A6) and (A7) results in

$$d(m_{j+1},q) \leqslant \cdots \leqslant d(m_k,q) \leqslant (1+\epsilon) d(n_{j+1}^0,q).$$

Thus the preliminary neighbors fulfill condition 2 of Sec. IV F for $i=j+1,\ldots,k$.

[1] H. Kantz and T. Schreiber, *Nonlinear Time Series Analysis* (Cambridge UP, Cambridge, 1997).

[2] H. D. I. Abarbanel, *Analysis of Observed Chaotic Data* (Springer-Verlag, New York, 1996).

[3] T. Schreiber, Int. J. Bifurcation Chaos Appl. Sci. Eng. **5**, 349 (1996).

[4] U. Parlitz, in *Nonlinear Modeling—Advanced Black-Box Techniques*, edited by J. A. K. Suykens and J. Vandewalle (Kluwer

Academic Publishers, Dordrecht, 1998).
[5] W. van de Water and P. Schram, Phys. Rev. A **37**, 3118 (1988).
[6] P. Grassberger, R. Hegger, H. Kantz, C. Schaffrath, and T. Schreiber, Chaos **3**, 127 (1993).
[7] J. McNames, *Proceedings of the International Workshop on Advanced Black-Box Techniques for Nonlinear Modeling, Belgium*, edited by J. A. K. Suykens and J. Vanderwalk (KU Leuven, Belgium, 1998), p. 112.
[8] R. Sedgewick, *Algorithms in C++*, 3rd ed. (Addison-Wesley, Reading, MA, 1998).
[9] D. Y. Cheng, A. Gersho, B. Ramamurthi, and Y. Shoham, in *Proceedings of the IEEE International Conference on Acoustics, Speech, Signal Processing, San Diego, 1984* (IEEE Press, Piscataway, NJ, 1984), p. 911.1.
[10] S. Berchtold, C. Böhm, D. A. Keim, and H. P. Kriegl, *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGARD Symposium on Principles of Database Systems, Tucson, Arizona, 1997* (ACM Press, New York, 1997), pp. 78–86.
[11] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu, J. Assoc. Comput. Mach. **45**, 891 (1998).
[12] G. Baier and M. Klein, Phys. Lett. A **151**, 281 (1990).
[13] G. Baier and S. Sable, Phys. Rev. E **51**, R2712 (1995).
[14] T. C. Halsey, M. H. Jensen, L. P. Kadanoff, I. Procaccia, and B. I. Shraiman, Phys. Rev. A **33**, 1141 (1986).
[15] S. Arya and D. Mount, http://www.cs.umd.edu/~mount/ANN
[16] V. N. Vapnik, *Statistical Learning Theory* (John Wiley & Sons, New York, 1998).
[17] V. Pestov, Inf. Process. Lett. **73**, 47 (2000).