Boosting k-Nearest Neighbor Queries Estimating Suitable Query Radii

Marcos R. Vieira[‡], Caetano Traina Jr.[§], Agma J.M. Traina[§], Adriano Arantes[§], Christos Faloutsos[†]

[‡]Department of Computer Science, George Mason University, Fairfax, VA - USA

[§]Computer Science Department, University of Sao Paulo at Sao Carlos, SP - Brazil

[†]Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA - USA

email: mrvieira@gmu.edu, {caetano|agma|arantes}@icmc.usp.br, christos@cs.cmu.edu

Abstract

This paper proposes novel and effective techniques to estimate a radius to answer k-nearest neighbor queries. The first technique targets datasets where it is possible to learn the distribution about the pairwise distances between the elements, generating a global estimation that applies to the whole dataset. The second technique targets datasets where the first technique cannot be employed, generating estimations that depend on where the query center is located. The proposed k-$NNF()$ algorithm combines both techniques, achieving remarkable speedups. Experiments performed on both real and synthetic datasets have shown that the proposed algorithm can accelerate k-NN queries more than 26 times compared with the incremental algorithm and spends half of the total time compared with the traditional k-$NN()$ algorithms.

1 Introduction

Complex data, such as genomic sequences and spatial and multimedia data, are searched mainly by similarity criteria, which motivated the development of algorithms to retrieve data based on similarity. Comparing two elements based on their similarity relies on a measurement, often calculated by a distance function $\delta\colon \mathbf{S} \times \mathbf{S} \rightarrow \mathbb{R}^+$, which compares a pair of elements of the data domain $\mathbf{S}$ and returns a numerical value that is smaller as the elements become more similar. A distance function is called a *metric* when for any elements $s_1, s_2, s_3 \in \mathbf{S}$ it satisfies the following properties: $\delta(s_1, s_1) = 0$ and $\delta(s_1, s_2) > 0, s_1 \neq s_2$ (non-negativeness); $\delta(s_1, s_2) = \delta(s_2, s_1)$ (symmetry); $\delta(s_1, s_3) \leq \delta(s_1, s_2) + \delta(s_2, s_3)$ (triangular inequality). Metrics are fundamental to create access methods to index a dataset $S \subset \mathbf{S}$, the so-called *Metric Access Methods* (MAM), which is true when $\mathbf{S}$ is a complex data domain. A MAM, such as the M-tree [2] and the Slim-tree [15], can accelerate similarity queries of complex data by orders of magnitude.

There are basically two types of similarity queries, both of which search a dataset $S \subset \mathbf{S}$ for elements $s_i \in S$ regarding their similarity to a query center $s_q \in \mathbf{S}$: the Similarity Range Query (RQ) and the k-Nearest Neighbor Query (kNNQ). The first retrieves every element in the dataset nearer than a given radius r_q to the query center. A range query example on a dataset S of genomic sequences is: **Q1:** *Choose the polypeptide chains that are different from the chain p by at most 5 codons.* The second retrieves the k elements in the dataset nearest to the query center. An example of a k-NN query on S is: **Q2:** *Choose the 10 polypeptide chains closer to polypeptide chains p.*

Due to the high computational cost to calculate distances between pairs of elements in complex domains, similarity queries often take advantage of index structures. Trees are the most common index structures for metric domains, where each node typically stores a data element called the representative, a subset of elements, and a covering radius, so that no element stored in the node or in any of its subtrees is farther from the representative than the covering radius. In non-leaf nodes, each element stored is the representative of a node of the next tree level. Index structures employ the triangular inequality property of metric domains to prune subtrees, using a limiting radius: a node whose representative is farther than the limiting radius added to its covering radius never contributes to the answer. An algorithm $Range(s_q, r_q)$ executes range queries using r_q as the limiting radius, thus the pruning ability of the *Range* algorithms using index structures is typically high. However, there is no *a priori* limiting radius to perform a k-NN query.

A k-$NN(s_q, k)$ algorithm starts collecting a list L of k elements, sorted by the distance $\delta(s_q, s_i)$ of each element $s_i \in S$ to the query center s_q. The limiting radius dynamically keeps track of the largest distance from the query center to the elements in L. Whenever a nearer element is found, it is included in the list, removing the farthest element and reducing the limiting radius accordingly. The limiting radius must start with a value larger than the maximum distance between any pair of elements in the dataset (or simply with infinity). Until at least k elements have been found, no pruning can be executed. Moreover, while r_L is larger than the final distance r_k of the true k-th element nearest to s_q, the false answers

included in L cannot be discarded. If a proper limiting radius can be set from the beginning, a significant reduction of distance calculations can be achieved. Therefore, the problem posed is : "*How to estimate a suitable radius r_k to execute a Range Query - RQ that returns the same elements of a k-NNQ?*"

This paper proposes two novel techniques to precisely estimate the final limiting radius r_f of a k-NN query, taking advantage of the intrinsic dimensionality of the dataset. The first technique targets datasets where the global data distribution can be obtained by estimating their intrinsic dimensionality, which is the large class of self-similar datasets. The second technique targets datasets where it is not possible to assume a global data distribution, therefore local estimations are generated for each query, depending on where the query center is located. Based on those estimates, we developed a new algorithm, called $k\text{-}NNF(s_q, k)$, which uses those estimations to accelerate k-NN queries. In the experiments performed, our technique decreased the number of distance calculations up to 37%, the number of disk accesses up to 18%, bringing an overwhelming gain in time, as it reduced the total time demanded by the query processing up to 26 times.

The remainder of this paper is structured as follows: Section 2 discusses the related works; Section 3 provides a short description of the Correlation Fractal Dimension concept, which is employed to estimate the intrinsic dimension of a dataset; Section 4 introduces the proposed algorithms and the mathematical support used to define the estimated radius r_k and the $k\text{-}NNF()$ algorithm; Section 5 presents the experiments results; and Section 6 gives the conclusions of this paper.

2 Related Work

The development of algorithms to answer similarity queries has motivated much research based on index structures. A common approach is the "branch-and-bound" where an indexing tree is traversed from the root, and at each step a heuristic is used to determine which branches can be pruned from the search and which one must be traversed next. One of the most influential algorithms in this category was proposed by Roussopoulos et al. [10] to find the k-nearest neighbors in multidimensional datasets stored in R-trees [4]. This algorithm has also been used in MAM to perform similarity searching following the "branch-and-bound" approach, which we call here the *"df"* (depth-first) $k\text{-}NN()$ algorithm. The heuristic of this algorithm involves choosing in a node the next subtree to be explored, which is the subtree closest to the query object. Thus, one priority queue is used for each node visited.

In [2], another algorithm is proposed that creates a unique global list (i.e. it is employed only one priority queue for the whole query) of nodes not yet traversed. When a new subtree must be traversed, it proceeds from the node closest to the s_q, regardless of its level in the tree. Using the globally closest node generates an algorithm considered better than the *df* $k\text{-}NN()$ algorithm, which is called the *"bf"* (best-first) $k\text{-}NN()$ algorithm.

Another approach uses incremental techniques to answer similarity queries, such as the work of Hjaltason and Samet [5], which retrieves the $(k+1)$ nearest neighbor after the first k have been found. This algorithm employs the triangular inequality property to delay the real distance calculations as much as possible, using a priority queue to keep track of which objects or subtrees must be analyzed next. The triangular inequality limits the minimum and the maximum distance allowable for each element s_i to s_q, using the distance from s_q to a node representative s_{rep}. A dynamic limiting radius can be derived by composing the minima and maxima distances of several nodes, even before performing the first real distance calculation between the s_q and a stored element. Metric trees, such as the Slim-tree, store the distance from s_{rep} to each subtree (or to the stored elements in leaf nodes). This algorithm delays pruning subtrees and objects until no improvement can be made to the minima and maxima limits, making its priority queue management very costly. The performance of the priority queue is highly affected by the dataset intrinsic dimensionality, as we show in the experiments in Section 5. We call this the *"inc"* $k\text{-}NN()$ algorithm, and together with the *df* and *bf*, this is one of the $k\text{-}NN()$ algorithms that we compare to our proposed algorithm.

Previous works trying to estimate a limiting radius for k-NN queries were presented in [12, 6, 9]. Their approaches are based on a probability density function to estimate a limiting radius, using a histogram of distances based on chosen pivots. The drawbacks are that it requires a costly construction of a histogram of the distribution of distances between the dataset objects and a set of pivots and it does not capture distinct distributions over different parts of the data space. Moreover, the histogram must be updated if the dataset is modified, as well as the number of pivots. Also, the pivots to be used in this approach need to be carefully chosen. Our approach differs from the previous works because it does not need sampling as the others do; it only uses the intrinsic dimensionality of the dataset. Moreover, by using the intrinsic dimensionality of the dataset instead of the number of attributes (even in spatial datasets), our approach avoids the problem of object dispersion in high-dimensional spaces [7], allowing it to deal with the varying density of elements in each query region.

3 Background

Uniformity and independence assumptions have long been discredited to model data distributions [1]. Real data overwhelmingly disobey these assumptions because they typically are skewed and have subtle dependencies between attributes, leading the majority of estimates and cost models to deliver inaccurate, pessimistic figures [8].

On the other side, some experimental evidence has shown that the distribution of distances between pairs of elements, in the majority of real datasets, presents a "fractal behavior" or self-similarity; the properties of parts of the dataset in a usable range of scales being similar to the properties of the whole dataset. In self-similar datasets the distribution of distances between elements follows power laws [11, 3] as follows: Given a set of N objects in a dataset with a distance function $\delta()$, the average number k of neighbors within a given distance r is proportional to r raised to $\mathcal{D}$, where $\mathcal{D}$ is the correlation fractal dimension of the dataset [3]. Thus, the pair-count $PC(r)$ of pairs of elements within distance r follows the power law:

$$PC(r) = K_p \cdot r^{\mathcal{D}} \quad , \tag{1}$$

where K_p is a proportionality constant.

Whenever a metric between pairs of elements is defined for a dataset, a graph depicting Equation 1 can be drawn, even when the dataset is not in a dimensional domain. For the majority of real datasets, this graph, called the *Distance Plot*, is plotted in log-log scales results in an almost straight line for a significant range of distances. The slope of the line in the Distance Plot is the exponent $\mathcal{D}$ in Equation 1, so it is called the *Distance Exponent* [14].

The results are independent of the base of the logarithm used (in this paper we use the natural logarithm $\log_e(x)$). It is interesting to note that $\mathcal{D}$ closely approximates the correlation fractal dimension of a dataset, and therefore its intrinsic dimensionality [11]. Hence, $\mathcal{D}$ can be seen as a measurement of the distribution of distances between elements, even for non-spatial datasets.

Figure 1 shows the distance plot of a dataset whose elements are the geographical coordinates [1] of streets and roads in Montgomery County, MD, USA (MGCounty). As can be seen, the plots are linear for the most requested range sizes (typically we are not interested in radius much smaller or larger than the typical distances involved in the dataset).

The Distance Exponent $\mathcal{D}$ of any dataset can be calculated as the slope of the line that best fits the resulting curve in the Distance Plot, as in Figure 1. Therefore, Equation 1 can be expressed as

$$\log(PC(r)) = \mathcal{D} \cdot \log(r) + K_d, \quad K_d = \log(K_p) \quad . \tag{2}$$

$\mathcal{D}$ has many interesting properties derived from the Correlation Fractal Dimension. The main property is

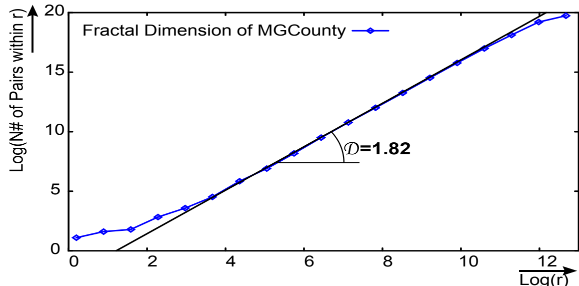

Figure 1. Distance Plot of the MGCounty.

that $\mathcal{D}$ is invariant to the size of the dataset, provided that a reasonable number of elements exists [3]. Therefore, the slopes of the lines corresponding to distinct datasets S from the same data domain $\mathbf{S}$ are always the same, regardless of the S cardinality. This property enables us to know $\mathcal{D}$ of a dataset even if the dataset had been updated.

4 How to estimate a suitable initial radius for k-NN queries

In this section we present the main contributions of this paper. Our goal is to take advantage of the intrinsic dimension of a dataset S to speed up $k\text{-}NN(s_q, k)$ queries estimating a final radius r_f, therefore the majority of the elements unlikely to pertain to the answer set are pruned right away, even before k elements have been inserted in L. The improved algorithm, which we call $k\text{-}NNF(s_q, k)$, performs three steps: **(1)** Estimates the final radius r_f, using a global estimate for r_k, **(2)** Performs a *radius-limited* $k\text{-}NN()$ algorithm using r_f, **(3)** Refines the search procedure, using a local estimate for r_f, if the required number of objects k was not retrieved.

The main ideas here are: (a) use the fractal dimension to estimate the ideal radius r_f; (b) over-estimating r_f in a systematic way; and (c) react when the estimate r_q is lower than the real final radius. The first and third steps require calculating respectively a global and a local estimate for the final radius r_f. In the next subsections we present techniques to calculate them. The second step can be performed by a new algorithm $kAndRange(s_q, k, r)$, defined as follows:

Definition 1 – *k-Nearest and Range Query:* Given an element $s_q \in \mathbf{S}$, a quantity k and a distance $r \in \mathbb{R}^+$, a "k-Nearest and Range algorithm" $kAndRange(s_q, k, r)$ retrieves at most k elements $s_i \in S \mid \delta(s_q, s_i) \leq r$, such that for every non-retrieved element s_j, $\delta(s_q, s_j) \geq \delta(s_q, s_i)$ for any retrieved element s_i.

The algorithm $kAndRange(s_q, k, r)$ can be implemented by modifying an existing $k\text{-}NN(s_q, k)$ algorithm changing the dynamic query radius initialization from infinity to r_f (obtained in the first step). The resulting algorithm answers a composite query, equivalent to the intersection of a $Range(s_q, r)$ and a $k\text{-}NN(s_q, k)$ using the same s_q. That is, considering the dataset S, the following queries are better expressed in relational algebra,

[1] This dataset is in the US Census Bureau Tiger format.

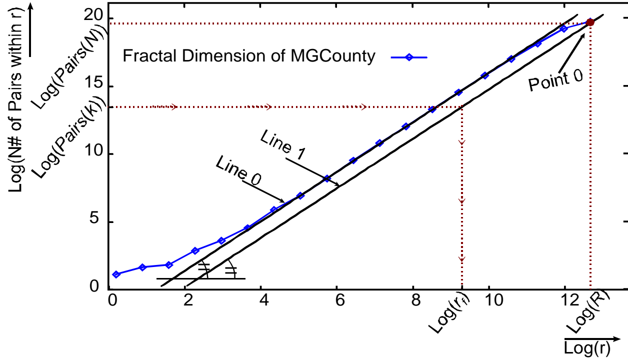

Figure 2. How to use the Distance Plot to obtain the global estimate $r_f = GEst(k)$.

as:

$$\sigma_{\left(Range(s_q,r_q)\wedge k-NN(s_q,k)\right)} S \Leftrightarrow$$
$$\sigma_{Range(s_q,r_q)} S \cap \sigma_{k-NN(s_q,k)} S \Leftrightarrow$$
$$\sigma_{kAndRange(s_q,k,r_q)} S \ .$$

For example, this algorithm alone could answer queries such as: **Q3:** *Select the 10 nearest restaurants not farther than a mile from here* – $kAndRange$(here, 10, 1mile). Notice that the algorithm will recover less than 10 restaurants if the range given is not sufficiently large.

4.1 Calculating a global estimate for the final query radius

The first step of the proposed k-$NNF(s_q, k)$ algorithm requires a method to estimate the final radius r_f of the k-NN query, so it can be used to call $kAndRange(s_q, k, r_f)$. It supposes that the elements in the dataset follow a mapped distribution regarding the distances between the elements (see Equation 1). Therefore, the resulting estimate is global for the whole dataset, independent of where the query is centered. The global estimate is defined as follows:

Definition 2 – *Global Estimate* $GEst(k)$: Given the number of neighbors k to be retrieved, $GEst(k)$ returns the estimated distance r_f of the equivalent range query that return k elements, based on the assumption that the dataset follows a self-similar distribution.

Figure 2 illustrates the main idea to obtain $GEst(k)$ using the Distance Plot of the dataset. To perform the estimate, an adequate line with slope $\mathcal{D}$, depicted as 'Line 1', must be chosen to convert the number of elements k into the corresponding estimated radius r_f.

The $GEst(k)$ method assumes that, as long as the general characteristics of the dataset regarding its distance distribution are preserved when the dataset is updated (i.e. the updates are performed randomly), the distance exponent remains the same. This means that the line best fitting the slope of the Distance Plot can go up or down as elements are inserted or deleted, but its slope is preserved. However, besides knowing the slope, one must define a particular line for the current dataset, using a known point in the graph. We propose using the total number of objects N currently in the dataset and the diameter R of the dataset. This pair of values defines a point in the graph, which we call 'Point 0'. The line of slope $\mathcal{D}$ including 'Point 0' estimates how to convert from "number of elements" to "radius". The number N is easily obtained (it is the number of elements in the dataset), and the diameter R can be estimated from the indexing structure, as the diameter of the root node of the indexing tree.

'Point 0' can be calculated through Equation 2. However, the equation uses the number of pairs $PC(r)$ within a distance r, and we are interested in the number of elements k involved. Therefore we must convert 'numbers of elements' into 'numbers of pairs' within a distance. The number of pairs in a subset of k elements (counting each pair only once) is $Pairs(k) = k(k-1)/2$. Thus, given a dataset with cardinality N, the number of pairs separated by distances less than the diameter of the dataset is $PC(R) = Pairs(N) = N(N-1)/2$, and a line specific to the dataset when a query is issued can be found considering 'Point 0' to be $\langle \log(R), \log(Pairs(N)) \rangle$ on the Distance Plot of the dataset. Using this line, the number of elements k that form pairs at a distance less or equal r can be estimated using $PC(r) = Pairs(k)$.

'Line 0' in Figure 2 is the one originally employed to calculate the intrinsic dimension $\mathcal{D}$ of the dataset (the line that best fits the Distance Plot). It approximates the average number of points (in log scale) over the full range of distances that occur in the dataset. Notice that real datasets usually have fewer distances with values near the diameter of the dataset, therefore the number of the largest distances increases at a slower pace than those of medium or small distances. This tendency explains why there is a typical flattening of pair-counting plots at large radius of real datasets, as shown in Figure 1.

Now let us consider 'Line 1' passing at 'Point 0'= $\langle \log(R), \log(Pairs(N)) \rangle$ (see Figure 2). As it represents the relationship between the radii and the number of element pairs within each radius in the whole dataset, it is adequate to estimate the final radius r_f of a k-NN query. The constant K_d from Equation 2 applied over 'Line 1' can be calculated as:

$$\begin{aligned} K_d &= \log(Pairs(N)) - \mathcal{D} \cdot \log(R) \\ &= \log\left(N(N-1)/2\right) - \mathcal{D} \cdot \log(R) \ . \end{aligned} \quad (3)$$

Notice that Equation 2 calculates the number of pairs in a set of k elements within a given distance. If we want to know the number of distances between a subset of k elements contained in S and the set of all N elements stored in the dataset S, then Equation 2 turns into:

$$Pairs(k) = N(k-1)/2 \ . \quad (4)$$

Combining Equation 3 with 4 and 2 we obtain:

$$\log(r_f) = [\log(PC(r)) - K_d]/\mathcal{D}$$

$$= [\log(N(k-1)/2) - \log(N(N-1)/2) + \mathcal{D} \cdot \log(R)]/\mathcal{D} \Rightarrow$$

$$r_f = R \cdot \exp([\log(k-1) - \log(N-1)]/\mathcal{D}) \quad (5)$$

Equation 5 shows how to calculate the global estimate for the final radius $r_f = GEst(k)$ of a k-nearest neighbor query.

The final radius r_f estimated by $GEst(k)$ is an approximation, as the exact radius of a query also depends on the local density of elements around the s_q. Therefore, if the query is centered where the density of elements is similar to or higher than the average density of the whole dataset, then calling $kAndRange(s_q, k, r_f)$ in Step 2 of algorithm $k\text{-}NNF(s_q, k)$ using r_f estimated by $GEst(k)$ answers the k-NN query with a much better performance than the traditional $k\text{-}NN(s_q, k)$ algorithms.

However, if r_f is under-estimated, then the $kAndRange(s_q, k, r_f)$ algorithm will return fewer elements than required (that is, a quantity $k' < k$ of elements), and another call to the $kAndRange()$ algorithm with a larger radius is required. As calling this algorithm a second time reduces the overall performance gain, it would help to slightly augment the estimate r_f to reduce the odds of having to call the $kAndRange()$ algorithm more than once. In fact, an augment is already embodied in Equation 5, when we estimate the size of the dataset by looking at the indexing structure: as the covering radius of the nodes of metric trees needs to guarantee that every element stored at each subtree is covered by that radius, they are always equal to or larger than the minimum required radius.

4.2 Calculating a local estimate for the final query radius

The third step of the proposed $k\text{-}NNF(s_q, k)$ algorithm is executed when r_f reveals to be under-estimated, despite the inflated diameter obtained from the root node of the metric tree. In this case, another search is required with a larger value $r'_f > r_f$. Calculating r'_f must take into account the local density of elements around the s_q, hence a local estimate $LEst()$, defined as follows, must be performed.

Definition 3 – *Local Estimate* $LEst()$: Knowing that the local data distribution around the s_q retrieves k' elements with a radius r_f, then $LEst(k, k', r_f)$ returns the local estimate r'_f required to retrieve the required number of neighbors k.

To develop the $LEst()$ algorithm, we take advantage of the Distance Plot again. We know that the first call to algorithm $kAndRange()$ using the estimated radius $r_f = GEst(k)$ returned an insufficient quantity k' of elements. Therefore, that first call was in fact a probe of the

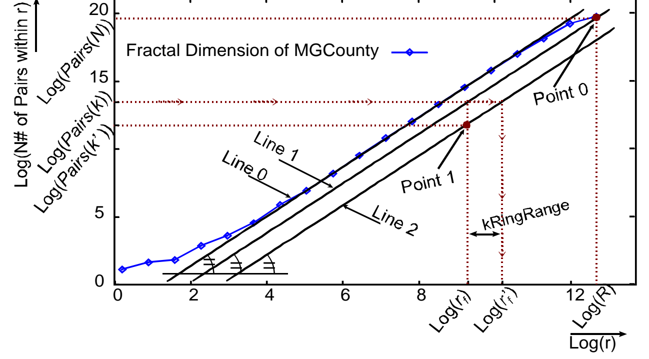

Figure 3. How to use the Distance Plot to obtain the local estimate $r_f = LEst(k, k', r_f)$.

space around the s_q, and the value k' can be used to estimate the local density of the dataset around the s_q. Let us reproduce Figure 2 in Figure 3, marking the point given by $\langle \log(r_f), \log(Pairs(k')) \rangle$ as 'Point 1'. This point corresponds to the actual amount of elements k' within distance r_f around the s_q. Now, let us draw another line with the same slope $\mathcal{D}$ containing this point, and call it 'Line 2'. 'Line 2' reflects the local density of the region where the query is centered, consequently it is useful to calculate the local estimate. The radius r'_f can be estimated by using equation 5, now using 'Line 2' and 'Point 1'. This leads to:

$$r'_f = r_f \cdot \exp([\log(k-1) - \log(k'-1)]/\mathcal{D}) \quad (6)$$

Equation 6 shows how to calculate the local estimate for the final radius $r'_f = LEst(k, k', r_f)$ of a k-NN query considering the local density of elements around the s_q.

Once the local estimate r'_f has been calculated, another incursion in the dataset must be performed to retrieve the additional $k - k'$ elements. This operation can be performed by another algorithm, defined as follows:

Definition 4 – *kRingRange*: Given an element $s_q \in \mathbf{S}$, the inner radius r_f, the outer radius r'_f and the maximum number of elements $k - k'$ still to be retrieved, the $kRingRange(s_q, k - k', r_f, r'_f)$ algorithm searches the dataset to find at most $k - k'$ elements s_i in the ring centered at s_q, so that $r_f < \delta(s_q, s_i) \le r'_f$.

The $kRingRange()$ algorithm is basically a $kAndRange()$ algorithm modified to prune not only elements and nodes that are outside the ball limited by the outer radius r'_f, but also the nodes and elements that are inside the ball defined by the inner radius r_f. The radius r'_f can be reduced whenever the required number of objects are found.

The first call to $LEst()$ and to the $kRingRange()$ algorithm retrieves k'' elements, where $k' \le k'' \le k$. It is not expected to have $k'' < k$, but it can. In this case, the point $\left\langle \log(r'_f), \log(Pairs(k'')) \right\rangle$ can be used to estimate another radius r''_f. This new estimate must then be used

in another call to the $kRingRange()$ algorithm, repeating this last step until the desired number of elements k are retrieved.

The k-$NNF()$ algorithm, shown as Algorithm 1, uses $kAndRange()$, $kRingRange()$, $GEst()$ and $LEst()$ to perform the same duty of a usual k-$NN()$ algorithm, receiving the same parameters and producing the same results, but with a better performance, as we show in the next section.

Algorithm 1 k-$NNF(s_q, k)$

Output: The list L of k pairs $\langle OId, \delta(s_{k_i}, s_q)\rangle$.
1: Obtain N as the number of elements in the dataset
2: Obtain R as the diameter of the dataset indexed
3: Clear list L
4: Set $r_f = GEst(k)$
5: Execute $KAndRange(s_q, k, r_f)$, store the answer in L, and set k' as the number of retrieved elements
6: **while** $k' \leq k$ **do**
7: Set $r'_f = LEst(k, k', r_f)$
8: Execute $kRingRange(s_q, r_f, r'_f, k - k')$, store the answer in L, and set k' to the number of elements in L
9: set r_f=r'_f
10: Return L

5 Experimental Results

To evaluate the effectiveness of the presented approach, we worked on a variety of synthetic and real datasets, experimenting with a variety of metrics, including vectorial and non-vectorial ones. As it is not possible to discuss all of them here, we selected three real and one synthetic datasets that represent and summarize the behavior of the proposed technique. The experimental results show that the estimates given by $GEst()$ and $LEst()$ are accurate, leading to the building of an efficient k-$NNF()$ algorithm.

The datasets are described as follows. The *CorelHisto* is a dataset of color histograms extracted from 68,040 images from the Corel Image Collection (http://kdd.ics.uci.edu). This is a real world dataset, where each element is a 32-dimensional vector (we consider it as a high-dimensional dataset). The *MetricHisto* is a real world dataset consisting of 40,000 metric histograms of medical gray-level images. It does not have a predefined embedded dimension, since the number of buckets (instead of bins in traditional histograms) varies from one image to another. Its elements are compared using the $MHD()$ metric function [13]. The *MGCounty* consists of a 2-dimensional coordinates dataset with 27,282 road intersections in Montgomery County, MD, Figure 4(a). The *Sierpinsky* is a synthetic dataset with 531,441 points of a Sierpinsky triangle, Figure 4(b). These are points from a perfect fractal, whose behavior is predictable, so it is useful to corroborate the correctness of the theoretical assumptions made in this work.

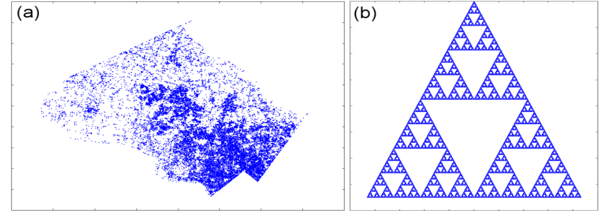

Figure 4. The two-dimensional vector datasets used in the experiments: (a) MGCounty and (b) Sierpinsky triangle.

Table 1. Summary of the datasets.

Dataset	$\mathcal{D}$	#Attrib.	# Objs	Metric
CorelHisto	4.97	32	68,040	Manhattan
MetricHisto	6.72	–	40,000	$MHD()$
MGCounty	1.82	2	27,282	Euclidean
Sierpinsky	1.56	2	531,441	Euclidean
SieDVar	1.56	(up to 256)	531,441	Euclidean

We also used several variations of the *Sierpinsky* dataset, adding up to 254 new attributes (*SieDVar*), calculated by polynomial expressions of the first two, to evaluate the behavior of the algorithms for datasets with varying embedded dimensionality and the same intrinsic dimensionality. Only 10% of each dataset were used to calculate its $\mathcal{D}$, as it is close to the one obtained using the whole dataset. Table 1 summarizes the main features of the datasets, including their intrinsic dimension $\mathcal{D}$, number of attributes (for the spatial datasets), number of elements, and the metric employed.

All algorithms were implemented in C++ using the GNU `gcc` compiler. The $kRingRange()$ and $kAndRange()$ algorithms were implemented following the approach given by the *bf* k-$NN()$ algorithm. We implemented the *inc* k-$NN()$ algorithm following the description and the best parameters given in [5]. The experiments were performed on an Intel Pentium IV 1.6GHz machine with 128 MB of RAM memory, using the Linux operating system.

Each measured point in a plot corresponds to 500 k-NN queries asking for the same number of neighbors at different query centers. The 500 query centers were sampled from the respective datasets, where half of them were removed from the dataset. Therefore 250 queries ask for neighbors from centers stored in the dataset, and 250 queries ask for neighbors from centers that are not in the dataset, but that correspond to centers likely to be used in real queries. The size of each disk page of the Slim-tree was 16KB for the *CorelHisto*, 4KB for the *MGCounty* and the *MetricHisto*, and 1KB for the *Sierpinsky* datasets. We used the default configuration parameters to build each Slim-tree. Tests performed with different page sizes obtained similar results.

5.1 Performance improvement

This section discusses the results of experiments comparing the proposed k-$NNF()$ algorithm with the most

representatives of the three existing classes (as explained in Section 2): the *df* k-$NN()$, the *bf* k-$NN()$ and the *inc* k-$NN()$ algorithms. Moreover, if the distance r_k from the query center to the farthest element in the final list of k nearest neighbors could be known in advance, then a *Range*() limited by r_k could be used in place of the k-$NN()$ algorithm because the processing of *Range*() is more efficient. Therefore, we also compared the k-$NN()$ algorithms to the standard *Range*() algorithm, using the limiting radius r_k previously discovered by one of the k-$NN()$ algorithms. This later test provides a theoretical lower bound of the best possible algorithm to find the nearest neighbors. All five algorithms were implemented using the Slim-tree as the indexing method. In this experiment, the number of neighbors in the k-NN queries varies from 5 to 100 in steps of 5 elements.

For each dataset, we measured the average number of distance calculations, the average number of disk accesses and the average total time (in seconds) required by each algorithm to perform each set of 500 k-NN queries. The number of distance calculations is important because comparing complex elements can be very costly. The reported number of disk accesses were obtained counting the number of requests generated by the algorithm, therefore issues regarding uses of cache memory were not taken into account. The objective of these measurements is to provide clues about the amount of data required by the algorithm, as the same index structure is used for every algorithm. However, the most important measurement is the total time, as it reports the global complexity of the algorithm. Thus, reducing the processing time is the ultimate objective of improving the performance of any algorithm.

Figure 5 shows the results of experiments on the four datasets. The first column shows the results from the *CorelHisto* dataset, the second from the *MetricHisto*, the third from the *MGCounty*, and the fourth column from the *Sierpinsky* dataset. The first row corresponds to the average number of disk accesses required to answer each query for the number of neighbors corresponding to the abscissas, the second row gives the average number of distance calculations, and the third row presents the average total time for 500 queries at each point in the plots.

Figure 5 shows that, as expected, no k-$NN()$ algorithm can perform better than the *Range*() algorithm, as it is the theoretical lower bound. It also shows that the average number of disk accesses and distance calculations required by the *inc* k-$NN()$ algorithm follows very closely those required by the theoretical best algorithm (the *Range*() one), as both plots are practically coincidental for every dataset. However, the *inc* k-$NN()$ algorithm achieves these results at the expenses of a heavy internal processing required to maintain the minima and maxima distances for the elements retrieved from the index structure. This processing turns the *inc* k-$NN()$ algorithm the slowest for every dataset, as shown in the third row of Figure 5.

Table 2. Total time (s) for k=10 and k=100.

Dataset	*df* k-$NN()$	*bf* k-$NN()$	k-$NNF()$	*Range*()	*inc* k-$NN()$
	k=10	k=10	k=10	k=10	k=10
CorelHisto	3.87	3.23	3.11	3.10	64.81
MetricHisto	16.89	15.97	15.03	14.98	206.25
MGCounty	0.48	0.47	0.34	0.28	7.13
	k=100	k=100	k=100	k=100	k=100
CorelHisto	9.78	9.03	7.19	5.69	137.87
MetricHisto	23.72	21.22	19.32	18.74	297.59
MGCounty	3.89	3.78	3.31	1.48	21.32

Table 3. *AvgOper* for all queues (k=20 and k=40).

Dataset	*df* k-$NN()$		*bf* k-$NN()$		k-$NNF()$		*inc* k-$NN()$	
	k=20	k=40	k=20	k=40	k=20	k=40	k=20	k=40
CorelHisto	173	183	173	183	110	117	2,601	2,856
MetricHisto	394	454	383	431	361	401	2,997	3,354
MGCounty	144	145	144	145	21	24	996	1,210
Sierpinsky	178	185	178	185	95	104	1,304	1,482

To better compare the other algorithms regarding time, Table 2 reproduces the values for k=10 and k=100 of the third row of Figure 5 for the first three datasets. The measurements show that the *inc* k-$NN()$ algorithm is up to 26 times slower than the others for the real world datasets (i.e., the *CorelHisto*, *MetricHisto* and *MGCounty* datasets). Therefore, the *inc* k-$NN()$ algorithm is valuable regarding the theoretical aspects of reducing the number of disk accesses and distance calculations, but it is impractical, as what really matters is the total time.

Figure 5 shows that, whereas the average number of disk accesses and distance calculations required by every algorithm present a sub-linear behavior for increasing number of neighbors, the average total time presents a slightly super-linear behavior. This is due to the increasing complexity to maintain the internal structures of the algorithms. In fact, the k-$NN()$ algorithms use a priority queue whose management has super-linear complexity regarding the number of elements managed. On the other side, the complexity of the proposed k-$NNF()$ algorithm is much smaller, reducing the processing time accordingly. Table 3 shows the total number of queue operations (add/remove elements to/from it) (*AvgOper*) averaged for 500 queries, for the four k-$NN()$ algorithms, using $k = 20$ and $k = 40$. As we see, the queue cost for the k-$NNF()$ algorithm is smaller than any other, contributing to making it the fast algorithm among the k-$NN()$ algorithms. Three aspects from this table enforce our finds: (1) the larger the embedded dimension of the dataset, the worse the performance of the *inc* k-$NN()$ algorithm; (2) the cost of the k-$NNF()$ queue is much smaller than the cost of the *inc* k-$NN()$ queue (up to 50 times smaller for the *AvgOper*); and (3) the k-$NNF()$ algorithm has smaller queue cost regarding any other algorithm.

Comparing the *df* and the *bf* k-$NN()$ algorithms in Table 2 and in the first and second rows of Figure 5, we can see that the *bf* k-$NN()$ is slightly better than the *df* k-$NN()$. However, comparing the *df* and the *bf* k-$NN()$ with the proposed k-$NNF()$, it can be seen that the k-$NNF()$ algorithm significantly reduces the three

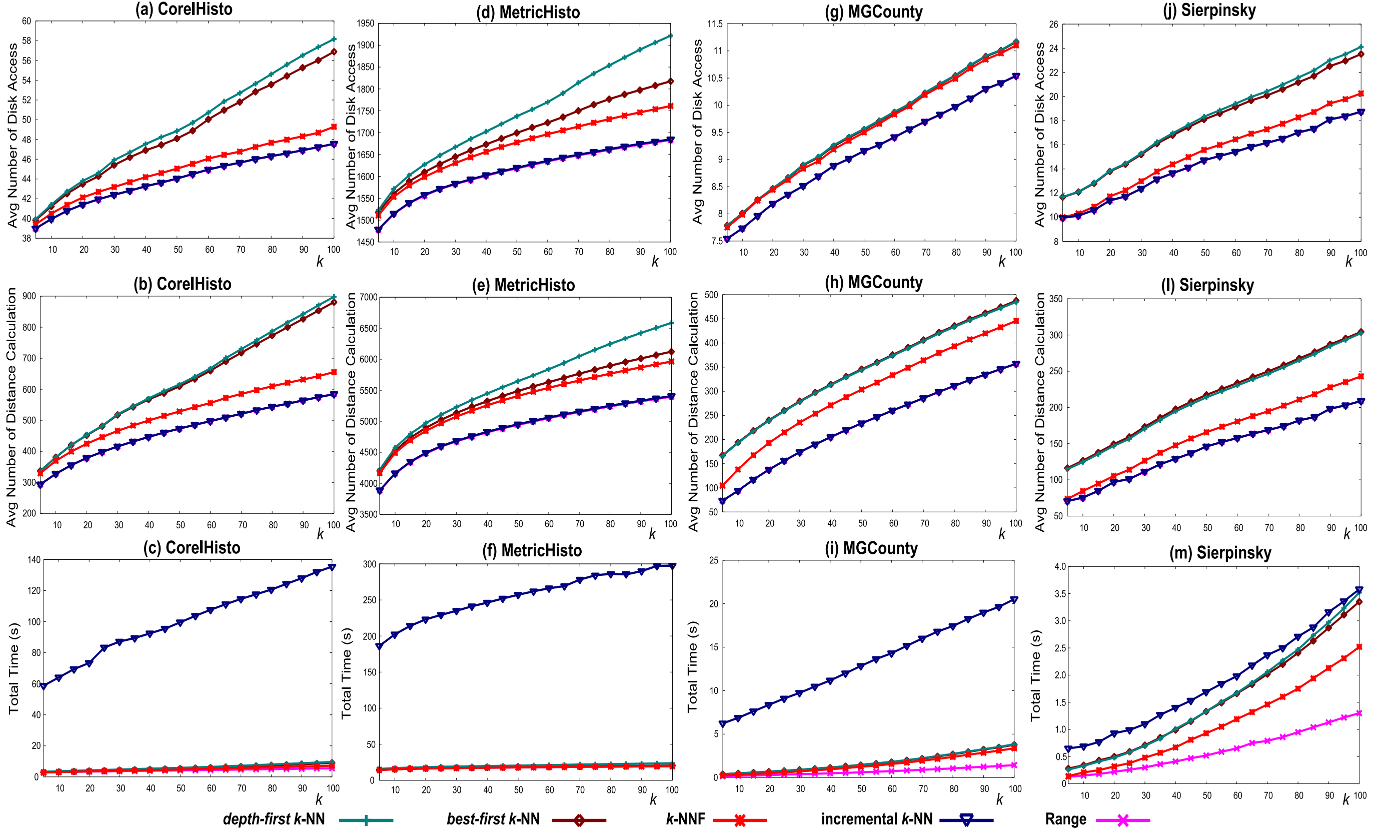

Figure 5. Comparing the proposed k-$NNF()$ with the other algorithms to answer 500 queries.

parameters measured (disk, distance, and time), for every dataset. The k-$NNF()$ algorithm decreases up to 37% (*MGCounty* and *Sierpinsky* - *df* k-NN) the need for distance calculations, decreases up to 18% the number of disk accesses (*CorelHisto* and *Sierpinsky* - *df* k-NN) and it spends half of total time than the traditional k-$NN()$ algorithms (*Sierpinsky*). Comparing the k-$NNF()$ algorithm with the *inc* k-$NN()$ algorithm (the best theoretical algorithm), it is up to 26 times faster than the later algorithm (*MGCounty*). The k-$NNF()$ algorithm achieves its best performance for the metric dataset *MetricHisto*, mainly considering that the *Range*() algorithm is a lower bound for improvements. As can be seen in Table 2, it can answer k-NN queries, spending at most 5% more time than the *Range*() algorithm, whereas its closest competitor requires at least 17% more time.

The *Range*() algorithm is the fastest because its internal structure does not require a priority queue to order by distance all subtrees to be visited, nor does it require the adjustment of the dynamic radius. As the k-$NNF()$ algorithm is based on the *kAndRange*() algorithm, it does need a priority queue, but it remains smaller than the others as a limiting query radius is available from the beginning. Excluding the *Range*() algorithm which requires a previous knowledge of the maximum radius, the proposed k-$NNF()$ algorithm is always the fastest one for every dataset evaluated.

5.2 Internal behavior of the k-$NNF()$ algorithm

In this section we present measurements about the internal behavior of the k-$NNF()$ algorithm, showing statistics about the estimates $GEst()$ and $LEst()$. Figure 6 shows the Distance Plot of the *MGCounty* dataset superimposed with the final radius r_f from the query center to its k-th nearest neighbor for the various values of k employed to generate the third column of plots from Figure 5 (with k varying from 5 to 100 in steps of 5 elements). Each point shown as a '□' represents the point $\langle \ln(r_f), \ln(Pairs(k)) \rangle$ in the log-log scale of the graph, which is used as parameters of the *kAndRange*() algorithms. It is clearly seen that the set of points are over a line passing close to $\langle \ln(R), \ln(Pairs(N)) \rangle$ ('Point 0' in Figure 3), the rightmost point of the Distance Plot used to obtain the fractal dimension of the dataset. Therefore, these points represent the global estimate $GEst(k)$ for each number of neighbors k asked in the queries.

Figure 6 also shows the average distance of r_f and the minimum/maximum radii for each value of k, measured from the set of 500 queries. As can be seen, the averages are roughly over the Distance Plot curve, confirming that it gives a suitable estimation of the final distance r_k. Moreover, it also shows that a sample of the dataset, such as the centers of the 500 queries, can be used to estimate

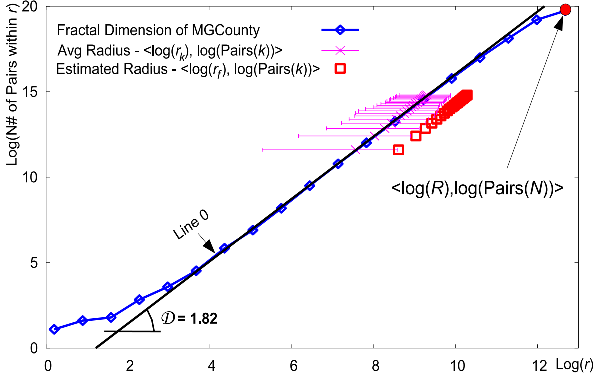

Figure 6. Global estimations for the distances from the query centers to their k-th NN and the average and minimum/maximum of the real distances measured, for the $MGCounty$ dataset.

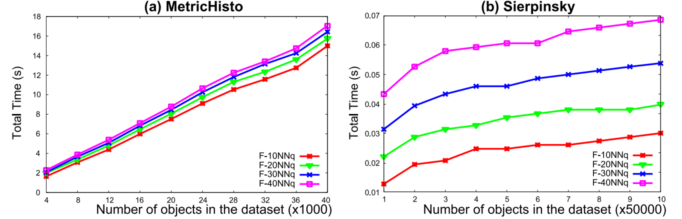

Figure 7. Scalability tests for k-$NNF()$.

the intrinsic dimension $\mathcal{D}$ of the dataset, as shown in the Figure 6. It also shows that the increase in the estimation of r_f (provided by the flattening of the curve and the super-estimation of the dataset radius) is enough to lead the number of required callings to $kRingRange()$ to a very small value, as the estimated values of r_f are always close to the maximum value r_k. Therefore, Figure 6 confirms that the global estimations $GEst()$ is accurate.

An thought-provoking point to check is determining when the $kRingRange()$ algorithm needs to be called, that is, when a local estimation $LEst()$ is required. To verify this point, we measured the number of times that $kRingRange()$ is called at each set of 500 queries using the same number k of required elements (step 8 of Algorithm 1). It turned out that the $kRingRange()$ algorithm was never called a second time. In fact, the algorithm is seldom called. Only for $k = 100$ the $kRingRange()$ was called 5 out of 500 queries for the $CorelHisto$ dataset, but was never called for the other datasets, showing that the $GEst()$ estimation is accurate. The fact that the $kRingRange()$ algorithm was never called twice confirms that the local estimation $LEst()$ is also very accurate.

5.3 Scalability and Dimensionality

In this section we present results of experiments performed to evaluate the behavior of the proposed k-$NNF()$ algorithm when varying the dataset size. To perform these experiments, we shuffled each dataset randomly and divided it in 10 parts of equal size. We created the index structure by inserting the first part and measured the first point of the plots. Thereafter the next part was inserted and the second point was taken, and so on, until having the measurements for the complete dataset. As before, each point in the plots corresponds to the average total time of 500 queries with different query centers. We performed experiments asking for 10, 20, 30, and 40 nearest neighbors (500 queries for each value of k). Figure 7 presents the plots for the average total time to answer 500 queries for the $MetricHisto$ (a) and $Sierpinsky$ (b) dataset. As can be seen, the results reveal an essentially linear behavior for every measured parameter. Figure 7(b) presents the plots for the $Sierpinsky$ dataset. As can be seen, the results reveal sub-linear behavior. The plots for average number of disk accesses and distance calculations follow a similar pattern (not shown here). The experiments with other datasets revealed that the general behavior is at least linear for high dimensional and non-dimensional (pure metric) datasets and is sub-linear for the low dimensional datasets.

We also performed experiments to evaluate the behavior of the proposed k-$NNF()$ algorithm when varying the embedded dimension. To evaluate these experiments, we generated several versions of the $SieDVar$ dataset, increasing the embedded dimension and keeping its intrinsic dimension constant. The extra dimensions were generated as linear combinations of the previous dimensions to increase the embedded dimension while keeping the fractal dimension unchanged. We created one index structure for each version, having the same height, node occupation and other parameters, keeping the same properties for every structure of each dataset, aiming at performing a fair comparison. As before, each point in the plots correspond to the average total time of 500 queries with different query centers, asking for 10 nearest neighbors. Figure 8 presents the plots for the various versions of the $SieDVar$ dataset, varying the embedded dimension from 2 (original dataset) to 256. As can be seen, the results reveal that our proposed technique works very well for datasets with high embedded dimensionality. On the other hand, the inc k-$NN()$ algorithm degenerates very fast when increasing the embedded dimensionality. From this experiment, we can conclude that as the dimensionality grows, the performance of the inc k-$NN()$ algorithm deteriorates. On the other hand, the k-$NNF()$ algorithm maintains its good performance when the dimensionality of the dataset increases, maintaining its position as the best of all algorithms.

6 Conclusions

This paper aims to develop techniques to explore the intrinsic dimensionality of a dataset, measured by its correlation fractal dimension, to estimate the final radius of

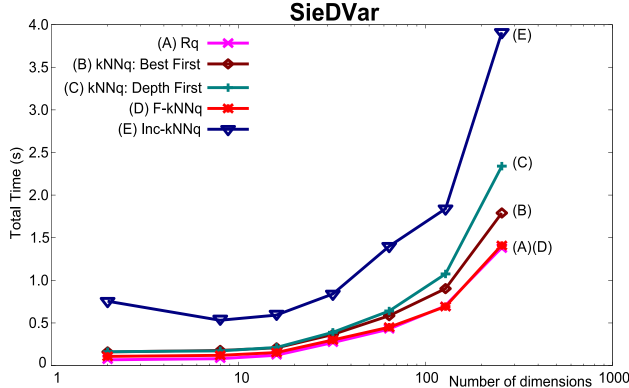

Figure 8. The embedded dimensionality test.

a k-nearest neighbor query to accelerate the query processing. Its main contributions are the following: **(1)** Two methods to estimate the final radius were proposed and implemented. The first one assumes the elements in the dataset following a known distribution (given by $\mathcal{D}$ obtained from the Distance Plot) regarding the distances between the elements and generates a global estimate $GEst()$. The second method takes into account the local density of elements around the query center, and generates a local estimate $LEst()$; **(2)** Using the two estimates provided by $GEst()$ and $LEst()$, the paper presents the new k-$NNF()$ algorithm, which improves processing k-NN queries over both spatial and metric datasets; **(3)** The two methods are independent of the underlying MAM employed; **(4)** The experiments compared the proposed k-$NNF()$ algorithm with the three most representative existing ones from their corresponding classes: the *df* k-$NN()$, the *bf* k-$NN()$ and the *inc* k-$NN()$ algorithms. We also compared k-$NNF()$ to the standard range query, using the previously-known limiting radius r_f. This later test provides a theoretical lower bound of the best possible algorithm to find the nearest neighbors. The experiments were performed on both real and synthetic datasets.

Although our approach requires more distance calculations and disk accesses than the best incremental algorithm, it requires very few calculations to estimate the final radius and few extra operations besides the element comparisons to perform the search procedure, as it requires smaller queues than the others to manage its subtrees. Therefore, the proposed k-$NNF()$ algorithm was the fastest for every dataset evaluated and required almost no extra memory to achieve such performance. The experiments conducted to evaluate the algorithms showed that the k-$NNF()$ algorithm can reduce at least one order of magnitude the time demanded to answer a query and is up to 26 times faster than the *inc* k-$NN()$ algorithm. Considering the non-incremental algorithms, our propose algorithm requires up to 37% less distance calculations, up to 18% less disk accesses, and spends half of the total time of its closest competitor. Besides presenting a real improvement in answering k-NN queries, the proposed algorithm is simple to implement, consumes very little additional memory and requires less computational power than the competing algorithms in the literature.

As additional research problems, we are interested in using the global and local estimates to improve clustering and classification algorithms, and also interested in employing the results of traditional k-NN and other similarity queries to calculate the intrinsic dimensionality $\mathcal{D}$ of a dataset.

Acknowledgement

This work has been supported by CNPq (Brazilian National Council for Research Supporting) and FAPESP (São Paulo State Research Foundation) grants, and a CAPES (Brazilian Federal Agency for Post-Graduate Education)/Fulbright Ph.D. fellowship for the first author.

References

[1] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *TODS*, 1984.

[2] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, 1997.

[3] C. Faloutsos, B. Seeger, A. J. M. Traina, and C. Traina Jr. Spatial join selectivity using power laws. In *SIGMOD*, 2000.

[4] A. Guttman. R-tree: A dynamic index structure for spatial searching. In *SIGMOD*, 1984.

[5] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *TODS*, 2003.

[6] L. Jin, N. Koudas, and C. Li. Nnh: Improving performance of nearest-neighbor searches using histograms. In *EDBT*, 2004.

[7] N. Katayama and S. Satoh. Distinctiveness-sensitive nearest-neighbor search for efficient similarity retrieval of multimedia information. In *ICDE*, 2001.

[8] F. Korn, B. Pagel, and C. Faloutsos. On the dimensionality curse and the self-similarity blessing. *TKDE*, 2001.

[9] C. A. Lang and A. K. Singh. Accelerating high-dimensional nearest neighbor queries. In *SSDBM*, 2002.

[10] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *SIGMOD*, 1995.

[11] M. Schroeder. *Fractals, Chaos, Power Laws.* 1991.

[12] M. Tasan and Z. M. Özsoyoglu. Improvements in distance-based indexing. In *SSDBM*, 2004.

[13] A. J. M. Traina, C. Traina Jr., J. Bueno, F. Chino, and P. M. Marques. Efficient content-based image retrieval through metric histograms. *WWWJ*, 2003.

[14] C. Traina Jr., A. J. M. Traina, and C. Faloutsos. Distance exponent: a new concept for selectivity estimation in metric trees. In *ICDE*, 2000.

[15] C. Traina Jr., A. J. M. Traina, C. Faloutsos, and B. Seeger. Fast indexing and visualization of metric datasets using slim-trees. *TKDE*, 2002.