Effective Proximity Retrieval by Ordering Permutations

Edgar Chavez, *Member*, *IEEE Computer Society*, Karina Figueroa, *Member*, *IEEE Computer Society*, and Gonzalo Navarro, *Member*, *IEEE Computer Society*

Abstract—We introduce a new probabilistic proximity search algorithm for range and K-nearest neighbor (K-NN) searching in both coordinate and metric spaces. Although there exist solutions for these problems, they boil down to a linear scan when the space is intrinsically high dimensional, as is the case in many pattern recognition tasks. This, for example, renders the K-NN approach to classification rather slow in large databases. Our novel idea is to predict closeness between elements according to how they order their distances toward a distinguished set of anchor objects. Each element in the space sorts the anchor objects from closest to farthest to it and the similarity between orders turns out to be an excellent predictor of the closeness between the corresponding elements. We present extensive experiments comparing our method against state-of-the-art exact and approximate techniques, both in synthetic and real, metric and nonmetric databases, measuring both CPU time and distance computations. The experiments demonstrate that our technique almost always improves upon the performance of alternative techniques, in some cases by a wide margin.

Index Terms—Similarity searching, metric spaces, indexing methods, information search and retrieval, pattern recognition.

1 Introduction

The classical Pattern Recognition process has three main stages: segmentation, feature extraction, and classification [29]. Segmentation consists of extracting the individual objects from the digitalized data. Feature extraction consists of mapping the digital objects onto a (usually high-dimensional) vector space, where each coordinate represents the degree of presence of a certain feature in the object. Classification consists of assigning each object to one out of a set of predefined classes of objects. This model encompasses concrete pattern recognition tasks such as speech recognition, speaker identification, signature matching, handwriting recognition, face recognition, biometric identification, etc. [22].

Feature extraction converts the original classification problem into a geometric problem. Objects in the same class tend to be spatially close if the features are selected properly. The most popular classification techniques, such as support vector machines, neural networks, or K-nearest neighbors (K-NN), are defined in terms of geometry. Among those, K-NN classifiers are attractive because the training is implicit.

The K-NN approach translates the problem of classification into a *proximity search* problem (find the K representative objects closest to a new given element) in a high-dimensional feature space. Unfortunately, current methods for proximity searching suffer from the so-called *curse of dimensionality* [16]: Any method for proximity searching, no matter how well it works in low dimensionalities, ends up scanning the whole set of objects in high dimensionalities. Dimensionality reduction techniques are effective and well-known, but they pose an extra overhead on the system when the data is *intrinsically* high dimensional and the classification accuracy will drop if the distances in the lower dimensional space are not well preserved. That is, the data will be misclassified when using a K-NN approach in a mapped space that distorts the original distances.

To avoid mapping onto a lower dimensional space, an abstract metric could be defined among objects (for example, the edit distance or dynamic time warping to match sequences) and can be transparently used as a black box in a K-NN classifier. In some cases, this is preferred over either mapping onto a vector space (to classify with a neural network) or defining a suitable kernel function (to classify with a kernel-based support vector machine).

In the so-called *metric spaces*, intrinsic dimensionality can be defined in many ways, for example, as the minimum dimensionality of a vector space onto which the metric space objects can be mapped without distorting much their pairwise distances. High-dimensional metric spaces have a concentrated histogram of distances and, just as on high-dimensional vector spaces, no proximity search algorithm can avoid comparing the query against all of the database.

Apart from classification, there are many other application areas for proximity searching: searching for similar objects in multimedia databases, searching for similar documents in information retrieval, searching for similar biological sequences in computational biology, data prediction, correction, or compression in signal processing, and so on. In all cases, the general model is that of a black box database of objects that can be preprocessed so as to

- *E. Chavez and K. Figueroa are with the Facultad de Ciencias Fisico Matematicas, Universidad Michoacana, Edificio B, Ciudad Universitaria, c.p. 58000, Morella, Michoacan, Mexico.*
  E-mail: {elchavez, karina}@fismat.umich.mx.
- *G. Navarro is with the Department of Computer Science, University of Chile, Blanco Encalada 2120, Santiago, Chile.*
  E-mail: gnavarro@dcc.uchile.cl.

answer proximity queries against new objects that are given later. The only tool to obtain information from the objects is the computation of their distance toward other objects. The curse of dimensionality shows up in all of these applications as well, in many cases rendering index-based methods as bad as a linear scan over the database or even worse.

Such a linear scan does not scale well when the set of objects to search is large or the distance function is computationally expensive. Different relaxations on the precision of the result have been proposed in order to obtain a computationally feasible solution in those cases. This is called *inexact proximity searching*, as opposed to the classical *exact proximity searching*. Inexact proximity searching is reasonable in many applications because the feature-extraction or the metric-space modelizations already involve an approximation to reality; hence, a second approximation at search time is usually acceptable.

In the literature, we basically find two alternatives for inexact proximity searching. The first one uses a distance relaxation parameter: It is ensured that the distance to the nearest neighbor answer they find is at most $1 + \varepsilon$ times the distance to the true nearest neighbor. This corresponds to approximation algorithms in the usual algorithmic sense and is considered in depth in [41], [16], [18]. A second alternative takes a probabilistic approach, ensuring that the answer of the algorithm is correct with high probability. This corresponds to probabilistic algorithms in the usual algorithmic sense. A generic method to convert exact into probabilistic algorithms is studied in [14], [10].

In this paper, we present a new probabilistic proximity search algorithm for metric spaces (which include vector spaces as a particular case). The central idea is to predict the closeness between any two objects in a metric space by comparing the way these two objects order their distances toward a set of anchor objects called *permutants*. The index does not store any actual distance but just permutations of the anchor objects as perceived by each database element.

We show that the similarity among permutations is a remarkably good predictor of the proximity among the corresponding objects. Thus, the database can be traversed from the permutation most similar to that of the query object to the least similar, and we expect to find most of the relevant answers early.

The probabilistic algorithm that results from traversing a given percentage of the database and returning the closest elements seen up to then is extremely efficient and outperforms any existing alternative that we are aware of. This is remarkable because there already exist very successful probabilistic techniques. We also tested our technique over nonmetric databases, using quasi-distances where the triangle inequality does not hold, and found that the retrieval effectiveness is comparable to that on metric databases.

2 Basic Concepts and Related Work

2.1 Basic Terminology

Formally, the proximity searching problem may be stated as follows: There is a universe $\mathbb{X}$ of *objects* and a nonnegative *distance function* $d : \mathbb{X} \times \mathbb{X} \longrightarrow \mathbb{R}^+$ defined among them. The distance satisfies the axioms that make the set a *metric space*: reflexivity $(d(x,x) = 0)$, strict positiveness $(x \neq y \Rightarrow d(x,y) > 0)$, symmetry $(d(x,y) = d(y,x))$, and triangle inequality $(d(x,z) \leq d(x,y) + d(y,z))$. This distance is assumed to be expensive to compute (think, for instance, of comparing two fingerprints). We have a finite *database* $\mathbb{U} \subseteq \mathbb{X}$, of size n, which is a subset of the universe of objects. The goal is to preprocess the database $\mathbb{U}$ to efficiently answer (that is, with as few distance computations as possible) *range queries* and K-nearest neighbor (K-NN) queries. Range queries are expressed as (q, r) (a point in $\mathbb{X}$ and a tolerance radius), which should retrieve all of the database points at distance r or less from q, that is, $\{u \in \mathbb{U}, d(u,q) \leq r\}$. On the other hand, K-nearest neighbor queries retrieve the K elements of $\mathbb{U}$ that are closest to q.

Most of the existing approaches to solving the search problem are *exact algorithms* that retrieve exactly the elements of $\mathbb{U}$, as specified above. In [16], [27], [38], [44], most of those approaches are surveyed and explained in detail. It is usually easier to design range search algorithms and then apply standard techniques to derive K-NN search algorithms from those.

2.2 Inexact Proximity Searching

In this work, we are interested in inexact algorithms, which relax the condition of delivering the exact solution. This relaxation uses an additional *precision* parameter ε to control how far away (in some sense) the outcome of the query can be from the correct result.

Approximation algorithms are surveyed in depth in [41]. An example is [4], which proposes a data structure for vector spaces under Minkowski metrics L_p. The structure, called the BBD-tree, is inspired by *kd*-trees and can be used to find "$(1 + \varepsilon)$ nearest neighbors": Instead of finding u such that $d(u,q) \leq d(v,q) \forall v \in \mathbb{U}$, they find u^* such that $d(u^*,q) \leq (1 + \varepsilon) d(v,q) \forall v \in \mathbb{U}$.

The essential idea behind this algorithm is to locate the query q in a cell (each leaf in the tree is associated with a cell in the decomposition). Every point inside that cell is processed so as to obtain the nearest neighbor u of q within the cell. The search continues with neighboring cells and stops when the radius of a ball centered at q and intersecting any cell not yet considered exceeds $d(q,u)/(1 + \varepsilon)$. The query time is $O(\lceil 1 + 6D/\varepsilon \rceil^D D \log n)$, where D is the dimensionality of the space.

Probabilistic algorithms have been proposed both for vector spaces [4], [43], [41], [23] and for general metric spaces [20], [18], [14], [10]. We survey a few of them.

In [43], the data structure is a standard *kd*-tree. The author uses "aggressive pruning" to improve the performance. The idea is to increase the number of branches pruned at the expense of losing some candidate points in the process. This is done in a controlled way, so the probability of success is always known. The data structure is useful for finding limited-radius nearest neighbors, that is, nearest neighbors within a fixed distance to the query.

In [23], the distance between two vectors is approximated by a convex combination of a *shape* measure of the vectors and their magnitudes. The shape measure has some resemblances to our technique as they sort the coordinates of vectors by increasing value. Nevertheless, our method applies to the more general metric spaces and does not use any equivalent to the magnitudes.

In [20], the author chooses a "training set" of queries and builds a data structure able to correctly answer only queries belonging to the training set. The idea is that this setup is enough to answer correctly, with high probability, an arbitrary query. Under some probabilistic assumptions on the distribution of the queries, it is shown that the probability of not finding the nearest neighbor is $O((\log n)^2/k)$, where k can be made arbitrarily large at the expense of $O(kn\alpha)$ space and $O(k\alpha \log n)$ expected search time. Here, α is the logarithm of the ratio between the farthest and the nearest pairs of points in the union of $\mathbb{U}$ and the training set.

In [10], the authors use a technique to obtain probabilistic algorithms that is relevant to this work. They use different techniques to *sort the database* according to some *promise value*. As they traverse the database in such an order, they obtain more and more relevant answers to the query. In other words, given a limited amount of work allowed, the algorithm finds each correct answer with some probability and it can improve the answer incrementally if more work is allowed. A good database ordering is one that obtains most of the relevant answers by traversing a small fraction of the database. Thus, the problem of finding a good probabilistic search algorithm translates into finding a good ordering of the database given a query q. Our contribution in this paper falls within this general approach.

Finally, there are approaches that combine approximation and probabilistic techniques, such as the PAC (probably approximately correct) method [17]. This is also the case in [14], which presents a general method based on stretching the triangle inequality.

2.3 Indexing

All metric space search algorithms rely on an *index*, that is, a data structure that maintains some information on the database in order to save some distance evaluations at search time. There exist two main types of data organizations [16], which we cover next.

2.3.1 Pivoting Schemes

A *pivot* is a distinguished database element whose distance to some other elements is precomputed and stored in an index. Imagine that we have precomputed $d(p,u)$ for some pivot p and every $u \in \mathbb{U}$. At search time, for a range query with radius r, we compute $d(p,q)$. Then, by the triangle inequality, $d(q,u) \geq |d(p,q) - d(p,u)|$, so that, if $|d(p,q) - d(p,u)| > r$, we know that $d(q,u) > r$, thus u can be filtered out without the need of computing distance $d(q,u)$.

The most basic pivoting scheme chooses k pivots $p_1 \dots p_k$ and computes all of the distances $d(p_i,u)$, $u \in \mathbb{U}$, into a table of kn entries. Then, at query time, all of the k distances $d(p_i,q)$ are computed and every element u such that $D(q,u) = \max_{i=1\dots k} |d(p_i,q) - d(p_i,u)| > r$ is discarded. Finally, q is compared against the elements not discarded.

As k grows, we have to pay more comparisons against pivots, but $D(q,u)$ becomes closer to $d(q,u)$ and more elements may be discarded. It can be shown that there is an optimum number of pivots k^*, which grows fast with dimensionality and becomes quickly unreachable because of memory limitations. In all but the easiest metric spaces, one simply uses as many pivots as memory permits. There exist many variations over the basic idea, including different ways to store the table of kn entries to reduce extra CPU time, for example, [13], [11], [32], [5], [12].

Several tree data structures are built on the same pivoting concept, for example, [42], [9], [30]. In most of them, a pivot p is chosen as the root of a tree and its subtrees correspond to ranges of $d(p,u)$ values being recursively structured. In some cases, the exact distances $d(p,u)$ are not stored; just the range can be inferred from the subtree the element u is in. Albeit this reduces the accuracy of the index, the tree usually takes $O(n)$ space instead of the $O(kn)$ needed with k pivots. Moreover, every internal node is a partial pivot (which knows distances to its subtree elements only), so we actually have many more pivots (albeit, local and with coarse data). Finally, the trees can be traversed using sublinear extra time.

Different tree variants arise according to the tree arities, the way the ranges of distances are chosen (trying to balance the tree or not), how local the pivots are (different nodes can share pivots, which may not belong to the subtree), the number of pivots per node, and so on. Very little is known about which is best. For example, the golden rule of preferring balanced trees, which works well for exact searching, becomes a poorer choice against unbalancing as the dimensionality increases. For very high-dimensional data, a good structure is almost a linked list (that is, a degenerate tree) [15]. Also, little is known about how to choose the pivots.

2.3.2 Local Partitioning Schemes

Another scheme builds on the idea of dividing the database into spatially compact groups, meaning that the elements in each group are close to each other. A representative is chosen from each group so that comparing q against the representative has a good chance of discarding the whole group without further comparisons. Usually, these schemes are hierarchical so that groups are recursively divided into subgroups.

Two main ways exist to define the groups. One can define "centers" with a covering radius so that all elements in its group are within the covering radius distance to the center, for example, [19]. If a group has center c and covering radius r_c, then, if $d(q,c) > r + r_c$, the whole group can be discarded. The geometric shape of this scheme corresponds to a ball centered around c.

In the second approach, for example, [8], [33], a set of centers is chosen and every other point is added to the group of its closest center. At query time, if q is closest to center c_i, and $d(q,c_j) - r > d(q,c_i) + r$, then we can discard the whole group of c_j. The geometric shape in this approach corresponds to a Dirichlet domain of the space (a generalization of the Voronoi diagram for metric spaces) without overlaps between groups.

3 An Effective Index Based on Ordering Permutations

Since the objects in the metric space are seen as black boxes from which we can only compute their distances toward other objects, all indexes in the literature are bound to store distance information. Actually, the most information an index can store is the $n \times n$ matrix of all the distances among objects in $\mathbb{U}$. This is actually what algorithm AESA [40], a pivot-based scheme, stores as its index. This makes

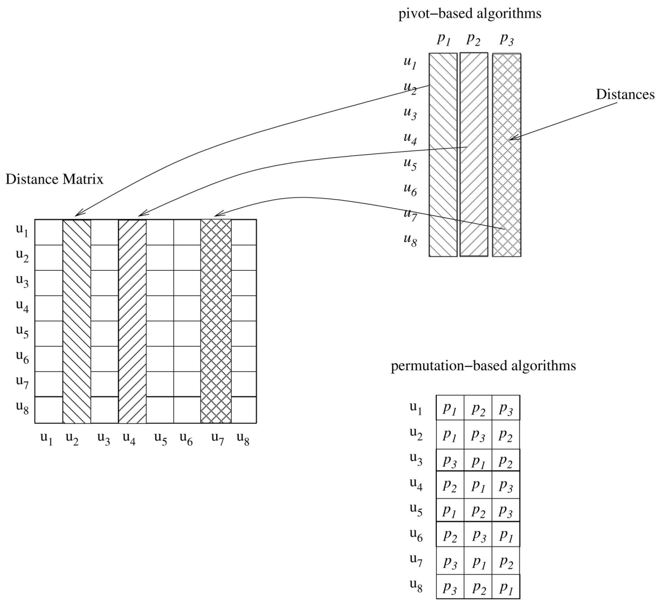


Fig. 1. (Left) The matrix of all distances in $\mathbb{U}$. (Top right) A pivot-based algorithm chooses some columns of the distance matrix. (Bottom right) Our algorithm only records the order of the pivots from the closest to the farthest element. Actually, only the permutation is stored, so, for example, the second row is stored as 1, 3, 2.

AESA an unbeatably exact algorithm, yet usually impractical because of its high storage consumption.

The design of metric space indexes can be regarded as a quest to store the most useful data from the distance matrix within bounded space. Pivot-based indexes store k columns from the full distance matrix, that is, for each element, they store its distances to k fixed pivots. Clustering algorithms store only some of the smallest distances in the matrix, that is, for each cluster center, they store the distances to the elements in that cluster. Some algorithms do not store the actual distances but just a range containing them so as to store more distances with less precision.

Within this framework, our approach can be stated as follows: *We choose k columns from the distance matrix and store, for each row, the order in which the columns are read to obtain the distances in increasing order.* Compared to a classical pivot-based scheme, we do not store the exact distances but just the order in which each database element sees the pivots, from the closest to the farthest element. That is, to each element, we associate a *permutation* of the k pivots. Fig. 1 illustrates.

Just as two close elements will have similar distances to pivots, close elements will see the pivots in similar order of closeness and, thus, will have similar permutations. A difference in the order between two permutations will hint that the corresponding elements are not too close to each other. However, those differences do not permit us to prove how far away from the query a database element is; thus, we will obtain a probabilistic algorithm.

3.1 Overview of Our Method

We need a bit of terminology. Let $\mathbb{P} \subseteq \mathbb{U}$ be a set of distinguished objects from the database called *permutants*. Each element of the space $x \in \mathbb{X}$ defines a *permutation* Π_x, where the elements of $\mathbb{P}$ are written in increasing order of distance to x. Ties are broken using any consistent order, for example, the order of the elements in $\mathbb{P}$.

Definition 1. *Let* $\mathbb{P} = \{p_1, p_2, \dots, p_k\}$ *and* $x \in \mathbb{X}$. *Then, we define* Π_x *as a permutation of* $(1 \dots k)$ *so that, for all* $1 \le i < k$, *it holds that either* $d(p_{\Pi_x(i)}, x) < d(p_{\Pi_x(i+1)}, x)$ *or* $d(p_{\Pi_x(i)}, x) = d(p_{\Pi_x(i+1)}, x)$ *and* $\Pi_x(i) < \Pi_x(i+1)$.

We are now ready to describe the indexing process, the index structure, and the search process.

3.1.1 Indexing

Our index will be just the permutations Π_u for every $u \in \mathbb{U}$, with respect to a set of permutants $\mathbb{P} = \{p_1, \dots, p_k\} \subseteq \mathbb{U}$.

The construction of the index is carried out as follows:

1. We choose a parameter k, which is the number of permutants to use. The larger k is, the more effective the index is, but it will need more space ($kn\lceil\log_2 k\rceil$ bits) and, also, sorting the database to traverse it in the desired order will be slower.
2. We choose $\mathbb{P} = \{p_1, \dots, p_k\}$, a set of k permutants, at random from $\mathbb{U}$. We will show in Section 4 that other selection heuristics of linear-time complexity make no difference in the effectiveness of the indexing algorithm.
3. For each $u \in \mathbb{U}$, we compute $d(u, p_i)$, for all $p_i \in \mathbb{P}$, and store permutation Π_u according to Definition 1.

The result is a table of n rows (one per database element) and k columns (one per permutant). Each cell needs $\lceil\log_2 k\rceil$ bits to store one permutation at each row. The indexing cost is kn distance computations plus $O(nk \log k)$ CPU time to sort all of the permutations.

3.1.2 Searching

At query time, we compute Π_q and traverse $\mathbb{U}$ in the order induced by Π_q. In this order, an element u will be smaller than an element v if Π_u is more similar to Π_q than Π_v. As we expect that elements with permutations more similar to Π_q will also be spatially closer to q, we will review them earlier.

The search is carried out as follows:

1. We compute $d(q, p_i)$ for all $p_i \in \mathbb{P}$ and compute permutation Π_q according to Definition 1.
2. Given a similarity measure S between permutations, we sort $\mathbb{U}$ according to $S(\Pi_u, \Pi_q)$ (those $u \in \mathbb{U}$ with smaller $S()$ value go first). Given that we will need just a (small) subset of the first elements after this sorting, we have used an incremental sorting method [36] which gives the elements in order as we need them. Other methods, such as a full QuickSort or BucketSort, were usually inferior.
3. We traverse the sorted elements $u \in \mathbb{U}$ and compute $d(u, q)$ for each such u. For range queries, we report any u such that $d(u, q) \le r$. For K-NN queries, we remember the K database elements that yielded the smallest $d(q, u)$ values so far.
4. We stop the scanning of $\mathbb{U}$ at some point and then deliver the result as obtained up to then, hoping that it will be close to the result that we would obtain by a full scan.

Say that we are willing to traverse $f \cdot n$ elements of $\mathbb{U}$. The total time complexity of the search process is k distance computations and $O(k \log k)$ CPU time for Step 1, $O(kn)$ CPU time to compute the $S()$ values (we see later that the measure S we use can be computed in $O(k)$ time) and $O(n + fn \log n)$ CPU time for the incremental sorting at Step 2, and, finally, $f \cdot n$ further distance computations for Step 3. This adds up $O(kn + fn \log n)$ CPU time and $k + fn$ distance computations. We tried some alternatives to avoid computing $S()$ for the whole database, but the result was not practical.

The stopping criterion deserves some discussion. The simplest is to scan a fraction $0 < f < 1$ of the database so that the amount of work is fixed beforehand and we have no control over the quality of the answer. Alternatively, we could like to fix an expected fraction $0 < p < 1$ of the correct answer retrieved. For K-NN queries, this can be obtained by previously building plots like those in the Appendix with a set of training queries. Those plots depend on the space but not on K. Later, given a K-NN query, we consider in the plot the points below $y = K/n \times 100$ percent in the y-axis. Now, we find the point x in the x-axis so that a fraction p of those points is to the left of x. This x value is the fraction of the database that we should traverse to obtain, on average, a fraction p of the K correct nearest neighbors. For range queries, the mechanism is similar, using a plot that, on the y-axis, gives the distance to the points found and $y = r$.

3.2 Measuring Similarity between Permutations

It remains to specify how we measure the difference between two permutations. We use Spearman Rho [24], denoted as $S_\rho(\Pi_q, \Pi_u)$, as our similarity measure: We sum the squares of differences in the relative positions of each element in both permutations. That is, for each $p_i \in \mathbb{P}$, we compute its position in Π_u and Π_q, namely, $\Pi_u^{-1}(i)$ and $\Pi_q^{-1}(i)$, and sum up the squares of the differences in the positions. A formal definition follows.

Definition 2. *Given permutations Π_u and Π_q of $(1 \ldots k)$, Spearman Rho is defined as*[1]

$$S_\rho(\Pi_u, \Pi_q) = \sum_{1 \le i \le k} \left(\Pi_u^{-1}(i) - \Pi_q^{-1}(i) \right)^2.$$

Let us give an example of $S_\rho(\Pi_q, \Pi_u)$. Let $\Pi_q = 6, 2, 3, 1, 4, 5$ be the permutation of the query and $\Pi_u = 3, 6, 2, 1, 5, 4$ be that of an element u. A particular element p_3 in permutation Π_u is found two positions off with respect to its position in Π_q. The differences between permutations are $1-2$, $2-3$, $3-1$, $4-4$, $5-6$, $6-5$, and the sum of their squares is $S_\rho(\Pi_q, \Pi_u) = 8$.

There are other similarity measures between permutations [24] such as Kendall Tau and Spearman Footrule. Kendall Tau is defined as follows: For every pair $p_i, p_j \in \mathbb{P}$, if p_i and p_j are in the same order in Π_u and Π_q, (that is, $\Pi_u^{-1}(i) < \Pi_u^{-1}(j) \Leftrightarrow \Pi_q^{-1}(i) < \Pi_q^{-1}(j)$), then $K_{p_i,p_j}(\Pi_u, \Pi_q) = 0$; otherwise, it is 1. Kendall Tau is given by $K(\Pi_u, \Pi_q) = \sum_{p_i,p_j \in \mathbb{P}} K_{p_i,p_j}(\Pi_u, \Pi_q)$, which is equal to the number of exchanges needed by a bubble sort to convert one permutation into the other. The Spearman Footrule between two permutations is

1. The actual definition in [24] corresponds to $\sqrt{S_\rho(\Pi_q, \Pi_u)}$ in our terminology. We omit the square root because it is monotonous and hence does not affect the ordering.

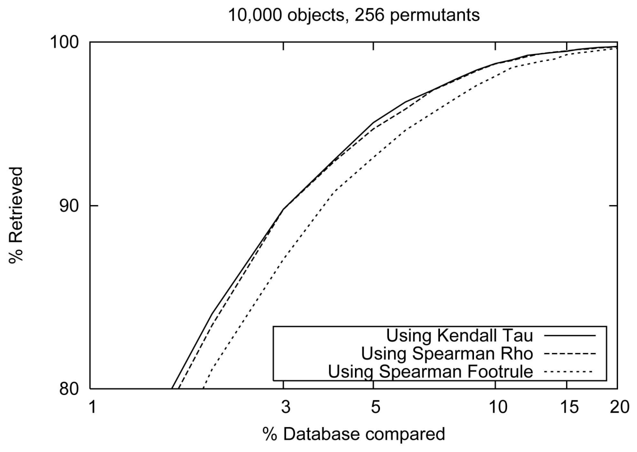


Fig. 2. Using different similarity measures between permutations (log scale). The space is a random uniformly distributed set of 10,000 points in the unitary cube of dimension 128 with euclidean distance. A total of 256 permutants were used.

$$F(\Pi_u, \Pi_q) = \sum_{1 \le i \le k} |\Pi_u^{-1}(i) - \Pi_q^{-1}(i)|.$$

In Fig. 2, we show that $F()$ is not as good as $S_\rho()$ for our purposes (similar results were obtained in other metric spaces). On the other hand, $K()$ performs similarly to $S_\rho()$, but it is more cumbersome to compute. Thus, we stick to Spearman Rho in the sequel.

We promised that S_ρ would be computable in linear time. According to Definition 2, this is easy if we store the *inverse* permutations Π_u^{-1} and Π_q^{-1}. As we prove next, it is enough to invert one of them to compute S_ρ in $O(k)$ time. Thus, we actually use Π_q^{-1} instead of Π_q.

Lemma 1. *Definition 2 is equivalent to*

$$S_\rho(\Pi_q, \Pi_u) = \sum_{1 \le j \le k} \left(j - \Pi_q^{-1}(\Pi_u(j)) \right)^2.$$

Proof. It is a matter of calling $j = \Pi_u^{-1}(i)$ and summing in different order. □

Algorithm 1 gives the complete pseudocode for range searching. It receives the query (q, r) and the fraction of the database $0 < f < 1$ to examine. The permutations Π_u, as well as the sets $\mathbb{U}$ and $\mathbb{P}$, are global variables. The database and the S_ρ values are stored as tuples $\langle u_i, S_\rho(\Pi_{u_i}, \Pi_q) \rangle$ in an array A, which is computed and then partially traversed to retrieve the (approximate) answer. For simplicity, we describe the algorithm as fully sorting A, not incrementally.

Algorithm 1 Sort-rangeQuery(q, r, f)
```
1:  INPUT: $q$ is a query and $r$ its radius, $f$ is the fraction of
        the database to traverse.
2:  OUTPUT: Reports a subset of those $u \in \mathbb{U}$ that are at a
        distance that is at most $r$ to $q$.
```

```
3:  Let $A[1,n]$ be an array of tuples and $\mathbb{U} = \{u_1, \ldots, u_n\}$
4:  Compute $\Pi_q^{-1}$
5:  for $i \leftarrow 1$ to $n$ do
6:      $A[i] \leftarrow \langle u_i, S_\rho(\Pi_{u_i}, \Pi_q)\rangle$
7:  end for
8:  SortIncreasing($A$) // by second component of tuples
9:  for $i \leftarrow 1$ to $f \cdot n$ do
10:     Let $A[i] = \langle u, s\rangle$
11:     if $d(q,u) \leq r$ then
12:         Report $u$
13:     end if
14: end for
```

4 Experimental Evaluation

In this section, we evaluate and compare the performance of our technique in different metric spaces, such as synthetic vectors on the unitary cube and clustered data (multivariate Gaussian distribution), as well as real-life databases like face images and text documents. We also tested the algorithm in nonmetric spaces, where the triangle inequality does not hold. All of the experiments reported excellent results for our method. They ran on an Intel Xeon workstation with 2.6 GHz CPU and 4 Gbytes of RAM with Red Hat Linux, running kernel 2.4.20-9.

4.1 Unitary Cube

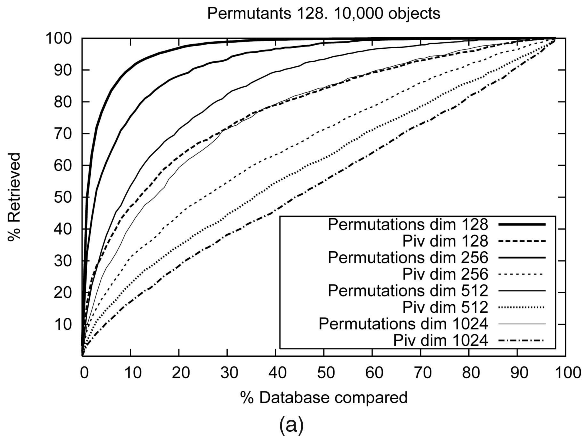
We made some experiments using uniformly distributed sets of 10,000 points in the unitary cube, in 128, 256, 512, and 1,024 dimensions, under euclidean distance. As we can precisely control the dimensionality of the space, we use this experiment to show how the predictive power of permutants varies with the dimensionality compared with other methods. We tested the range queries with a search radius that retrieved, on average, 0.05 percent of the database (that is, five points). We emphasize that no exact algorithm can avoid a linear scan of the database when we go over dimensionality 30 with uniformly distributed points; only probabilistic algorithms work.

We considered $k = 128$ and $k = 256$ permutants in our experiments. We compare our technique with a standard pivot-based method using the same amount k of pivots, even though this represents at least four times the memory we use for our algorithm. If we used the same amount of memory for the two algorithms, the comparison would be even more favorable to us.

The pivot-based probabilistic alternative we tested [10] calculates, for each database element u, estimate $L_\infty(q,u) = \max_{p \in \mathbb{P}} |d(q,p) - d(p,u)|$. The database is then sorted by increasing L_∞ value and compared against the query in this order.

Fig. 3 shows the comparison. The x-axis represents the percentage of the database examined and the y-axis is the percentage of the actual answer that was retrieved (this estimates the probability of returning a given answer element).

Retrieving 90 percent of the answer is good enough for most proximity searching applications. With 128 pivots, in dimensionality 128, 60 percent of the database must be examined to retrieve 90 percent of the results. For our permutation-based algorithm, with 128 permutants, we must examine only 10 percent of the database to retrieve 90 percent of the outcome. This raises to 99 percent if we use 256 permutants. With 256 pivots instead, one needs to compare 85 percent of the database to retrieve 99 percent of the answers.



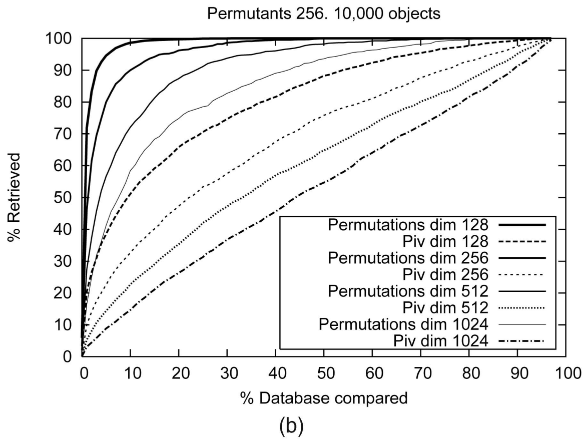


Fig. 3. Performance of our versus pivot-based probabilistic algorithms in different dimensionalities. In (a), we use 128 pivots/permutants and, in (b), 256. Series with the word piv refers to the standard pivot-based algorithm.

In general, we observe that, as the dimensionality grows, a larger fraction of the database must be examined to obtain a given fraction of the result. This observation is true for the pivot-based algorithm, as well as for ours. Nevertheless, the pivot-based algorithm is more affected by dimensionality than ours. Note that an algorithm that traverses the database in random order would achieve a straight line from the bottom-left to the top-right corner, that is, it needs to examine 90 percent of the database to obtain 90 percent of the answer. It can be seen that pivot-based algorithms actually behave almost randomly on very high dimensionalities.

Note that, in this synthetic data, we may be using more permutants than space coordinates. Since the permutation similarity is more expensive to compute than plain euclidean distance, this may seem nonsensical. We remark that this experiment is just to demonstrate the performance of the technique in terms of distance computations. Real data may have thousands of coordinates or no coordinates at all. We include real CPU times for all the other metric spaces that follow.

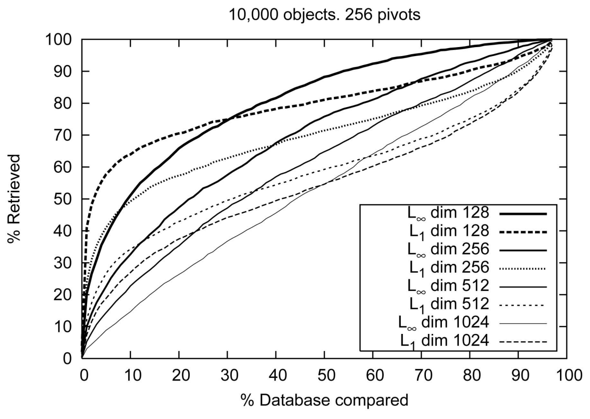


Fig. 4. Comparison between the L_1 and L_∞ Minkowski metrics to sort the database with pivot-based algorithms using 256 pivots.

One might wonder whether the L_∞ distance used by the pivot-based probabilistic algorithm is a good predictor. Although there are good reasons to use L_∞ [10], one can also argue in favor of L_1: AESA, the best exact algorithm [40], uses the L_1 metric as the oracle to select the next best candidates for pruning the database, that is, $L_1(q,u) = \sum_{p\in\mathbb{P}} |d(q,p) - d(p,u)|$. In Fig. 4, we show the results for the test for the L_1 distance to sort the database for the probabilistic algorithm based on pivots versus the L_∞ choice used above. It can be seen that the results are mixed. In the first part (for example, scanning less than 20 percent of the database in dimensionality 128), distance L_1 retrieves a larger percent of the database compared to L_∞. Nevertheless, once a turning point is reached, the result is reversed. The same behavior is observed in all of the dimensionalities considered. We emphasize that, anyway, the results are very far from what we obtain with our new technique.

In the Appendix, we display the power of the sorting methods using clouds of points. These show how often our technique put nearest neighbors in the first positions.

4.2 Gaussian Spaces

Uniformly distributed data is full dimensional. Real data sets behave more like clustered data, which is easier to index. We tested our algorithm on a Gaussian space. The data was generated for a 1,024-dimensional space $[0,1]^{1024}$ with 10,000 points obtained from a multivariate Gaussian distribution with 32 clusters (centers). The variance of the center distribution was 0.09 and the variance inside the clusters was 0.01.

Fig. 5 shows experiments of the CPU time needed for retrieving the nearest neighbor using 32 and 128 pivots/permutants. Notice that the ordering using permutations retrieves 100 percent of the answer faster than the others. In Fig. 5a, using 32 pivots/permutants, ordering using permutations retrieves 100 percent of the answer in just 0.03 sec, while the others require 0.17 sec.

4.3 Face Recognition

In many real-world scenarios, objects are modeled as very high-dimensional feature vectors. Spatial access techniques cannot be used efficiently in this case due to the curse of dimensionality. An alternative is to work

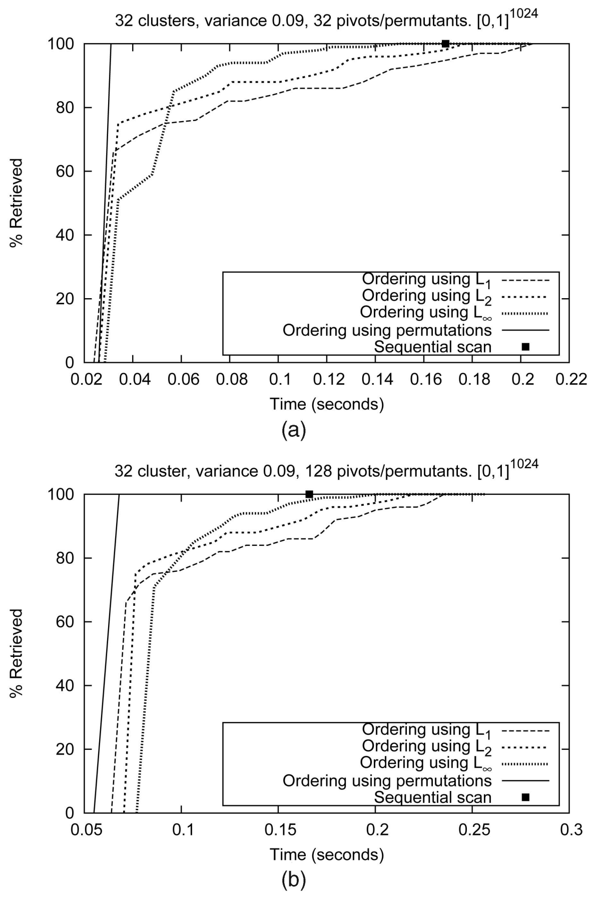


Fig. 5. Retrieving the nearest neighbor on a 1,024-dimensional Gaussian space with 32 clusters, using (a) 32 and (b) 128 pivots/permutants. We show the retrieval percentage versus the total time to obtain the results.

without coordinates, using the distance just as a black box, that is, resorting to the metric space model. Nevertheless, in several cases, the resulting intrinsic dimensionality is still very high and no exact search method can avoid an exhaustive scan of the database.

In this section, we consider the FERET database [37], which consists of 762 gray-scale frontal face images of 254 different people (3 images per person). The pictures are of 128×128 pixels, that is, each face is represented by 16,384 features. The query set has 254 images (1 image per person). To speed up searches, the vectors were transformed by eigenspace methods which project the input faces onto a 761-components (coordinates) space where the recognition is carried out.

We consider K-NN search, as this is the most frequent query in this application. For the probabilistic algorithms, we measure the number of distance computations performed (averaged over all the queries) until the algorithms obtain the correct K-NN. We used all 254 queries for each K value tested.

Since the size of the database allows it, we included AESA [40] in the comparison as it is considered a baseline to compare exact searching algorithms. AESA uses the entire distance matrix to answer queries and it is the best exact algorithm. As the distance is euclidean, we also experiment with a kd-tree [7] as an exact search method that attempts to reduce CPU time.

Figs. 6a and 6b show the results, using 64 permutants. It can be seen that the best exact technique (AESA) requires scanning 30-40 percent of the database to find the nearest neighbor, and this quickly raises to 80-90 percent for larger K. Kd-trees need 50 percent to find the nearest neighbor. Our technique performs better, scanning around 10 percent of the database on average to find the nearest neighbor and 30-40 percent for 20 nearest neighbors. For the probabilistic algorithm based on pivots, we chose the L_1 distance to sort the database. It requires traversing a larger fraction of the database to achieve the same result of permutants (40-50 percent for $K = 20$ neighbors). The results for L_∞ were not included as they are worse than for L_1.

Figs. 6c and 6d show real CPU times. It can be seen that, although permutations pose a CPU time overhead higher than pivots, the result is still advantageous in terms of CPU time. (Note that AESA is more expensive in practice than a sequential scan.)

Fig. 7 displays the results in a form more similar to previous plots. We show the percentage of queries successfully solved (that is, all of their K-NNs are found) after traversing a given percentage of the database. We also display the *relative error* ratio between the distance to the Kth nearest neighbor found divided by the distance to the true Kth nearest neighbor (computed only over the unsuccessful queries). It can be seen that, even when the algorithm fails to find the true answer, the approximation it finds is rather good.

Again, in the Appendix, we display the power of the sorting methods for this database.

4.4 Documents

A central problem in information retrieval consists of finding documents relevant to a given query. The relevance is measured using a specialized distance definition. Documents are represented as unitary vectors, where every coordinate corresponds to a term and the value of a document vector along each coordinate is proportional to the weight of the term in that document. The number of different terms in a collection is on the order of hundreds of thousands, resulting in a very high-dimensional vector space with the usual dimensionality curse problems. The distance between two documents can be taken as the angle between their representing vectors (the cosine of this angle is a similarity measure heavily used in information retrieval [6]).

We used a subset of collection TREC-3 [26] to compare the performance of our approach against the best previous results using probabilistic algorithms [10]. The database consists of 24,960 documents. We averaged 1,000 range queries chosen at random, with a radius retrieving, on average, 0.035 percent of the database (nine documents). No exact algorithm performs well in this setup: Even AESA needs to compare the query against 60 percent of the database to solve this query.

The results can be seen in Fig. 8a, using 128 pivots or permutants. Permutations quickly reach a good percentage of retrieval: We review just 2 percent of the database to retrieve 95 percent of the outcome, while the classical pivot-based algorithm (that is, using L_∞ ordering) needs to

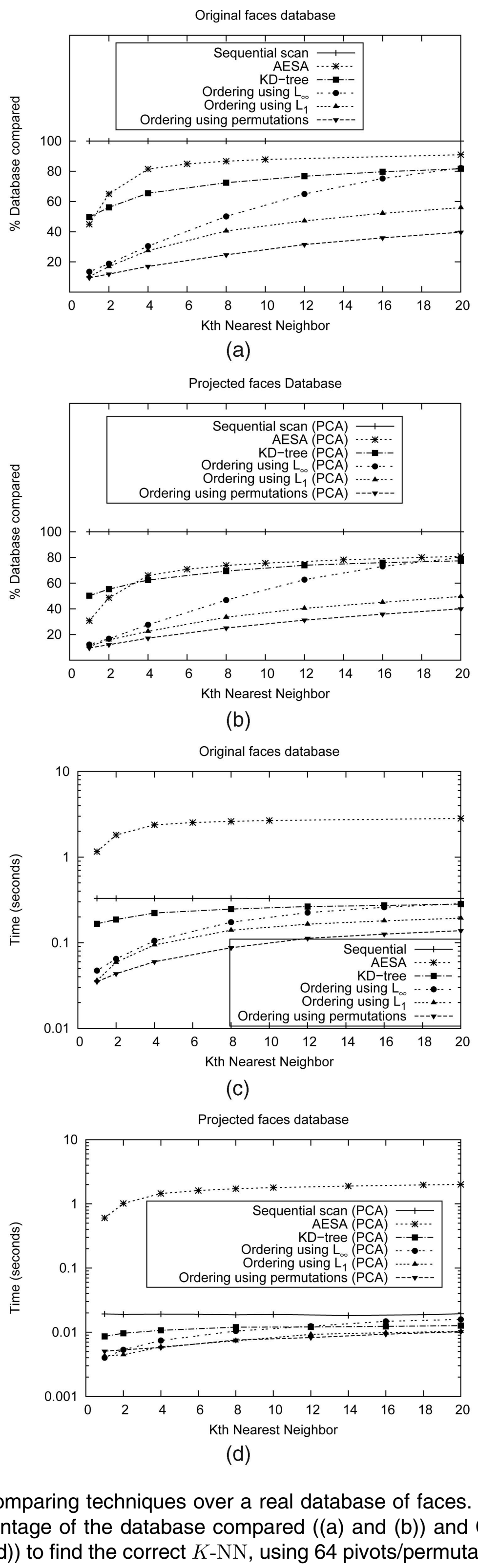
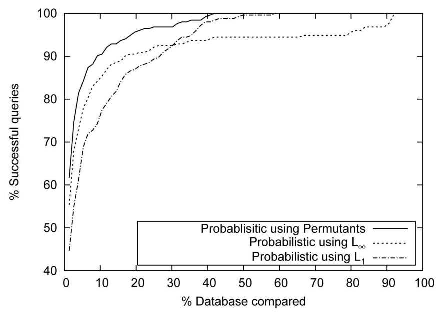
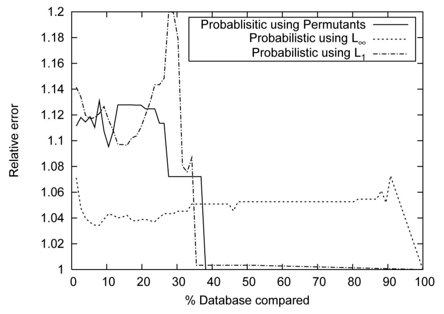


Fig. 6. Comparing techniques over a real database of faces. We show the percentage of the database compared ((a) and (b)) and CPU time ((c) and (d)) to find the correct K-NN, using 64 pivots/permutants. In (a) and (c), we work with the original space; in (b) and (d), we work with the projected space.

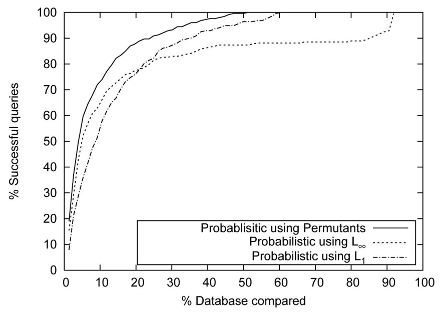
review almost 20 percent of the database to achieve the same retrieval performance. A pivot-based algorithm using L_1 (not tried before as far as we know) performs almost as well as permutations. Finally, in [10], a method called *Dynamic Beta* is proposed, which needs to review about



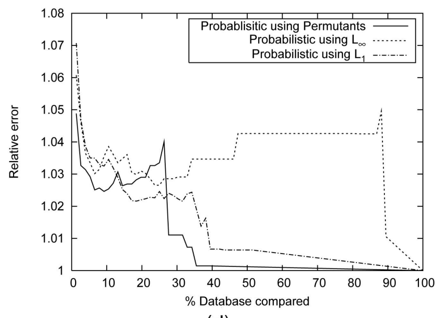
(a)



(b)



(c)



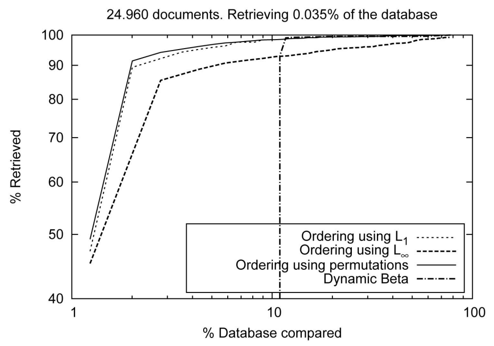
(d)

Fig. 7. Comparing techniques over a real database of faces. In (a) and (b), we use $K = 2$ and, in (c) and (d), we use $K = 4$. In (a) and (c), we show the percentage of queries where all the K-NN are correctly found. In (b) and (d), we show the relative error for those queries that do not find all the correct neighbors.

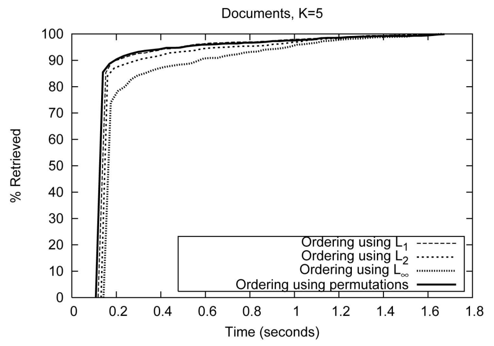
10 percent of the database to reach the same retrieval performance. After paying that 10 percent, *Dynamic Beta* by far surpasses the pivot-based method and, from then on, it becomes similar to permutations.

Fig. 8b shows the result of a 5-NN query, this time focusing on CPU times. Again, using permutations is (slightly) faster than the others.

We again display the power of the sorting methods on this database using clouds of points in the Appendix.



(a)



(b)

Fig. 8. Comparing our technique with others in a real database of documents. (a) Retrieval performance and (b) the CPU time compared against retrieval performance for all of the probabilistic algorithms.

4.5 Nonmetric Databases

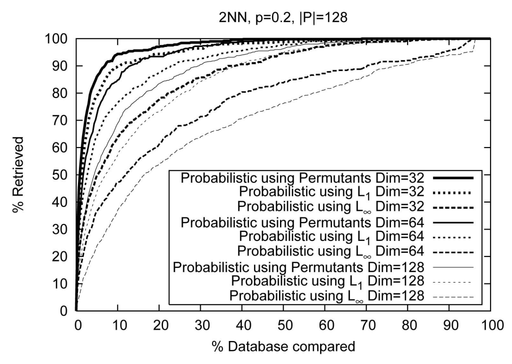
There are several real-life applications where similarity searching has to be carried out over a space that is not even metric, that is, where the triangle inequality does not hold. In this case, exact proximity search algorithms are useless in general as there is no way to prove that an element is sufficiently far away from the query q. A probabilistic algorithm instead has a chance of still proposing an appealing order to traverse the database. A variation of this idea, forging a monotonous transformation of the database, is indeed used in [39] as a good alternative to search in nonmetric databases.

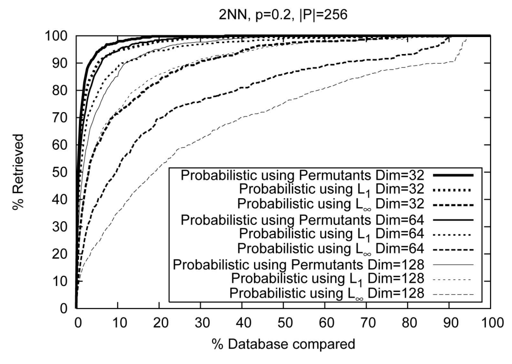
In particular, our probabilistic algorithm does not make use of the triangle inequality as it never discards an element; it just hints which are the most promising candidates to consider first. As such, it can be used on nonmetric databases.

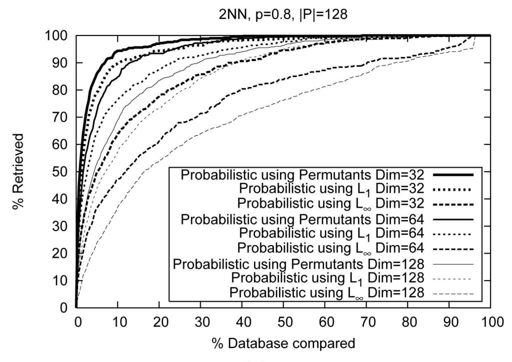
We apply our K-NN algorithm over a couple of nonmetric spaces in order to demonstrate its suitability. The first space is a synthetic uniform vector space, just as those in Section 4.1, using, instead of euclidean distance, a so-called *fractional norm* L_p with $0 < p < 1$:

$$L_p((x_1,\ldots,x_D),(y_1,\ldots,y_D)) = \left(\sum_{1\le i\le D} |x_i - y_i|^p\right)^{\frac{1}{p}}.$$

Fractional norms are sometimes preferred over the usual Minkowski norms, L_1, L_2, or L_∞, because they lead to lower intrinsic dimensionality [2], [1], [21], [28]. (Please do not confuse this norm, which is used as the d distance in the







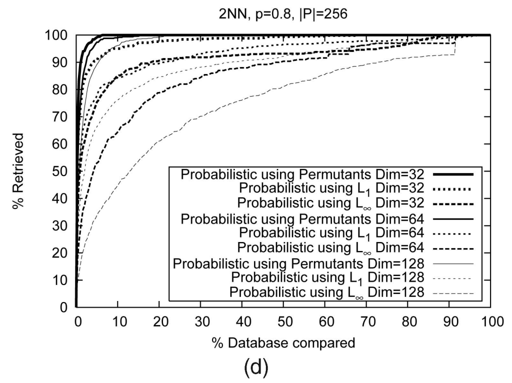


Fig. 9. Comparing our technique with others in uniformly distributed vector spaces using L_p distance (nonmetric, $p < 1$) to retrieve two nearest neighbors. In (a) and (b), $p = 0.2$ and, in (c) and (d), $p = 0.8$; we used 128 pivots on (a) and (c) and 256 on (b) and (d).

metric space, with the L_1 and L_∞ norms explained in Section 4.1 to sort the database. These are independent.)

Fig. 9 compares the performance of our ordering based on permutations with those based on L_1 and L_∞, as in

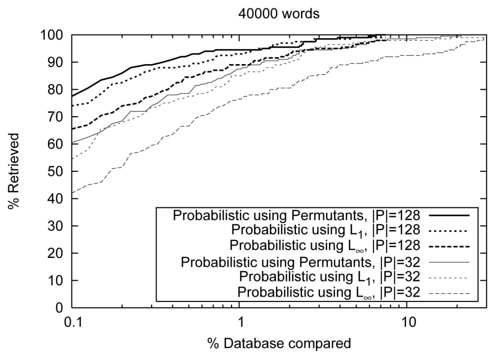


Fig. 10. Comparing our technique with others in a space of strings using NED. Range search with radius 1.

previous sections. It can be seen that permutations achieve the best result, followed by L_1. The problem is easier as p grows and the space gets closer to be metric.

The second space is that of sequences using *normalized edit distance (NED)* [31], [3]. The usual edit distance (which is a metric) favors short sequences over long ones, given the same fraction of similarity between the two sequences. The NED counterweights this bias by dividing the cost of a sequence of operations by the length of that sequence. The result is no longer a metric, but it works better in several applications.

Fig. 10 shows the results over 40,000 words from a dictionary using this distance for a range search with radius 1. In this case, the permutations and the L_1 orderings yield similar results, superior to those of L_∞.

4.6 Selecting Permutants

Permutants are central to our method. Hence, it is worthwhile to investigate the role of permutant selection. We tested heuristics based on selecting permutants with minimum or maximum Spearman Rho in the set: We start with a set with only one element and the next permutant will be selected minimizing (or maximizing) the sum $p = \min_{u_i \in \mathbb{U}} \sum_{p_j \in \mathbb{P}} S_p(u_i, p_j)$. This type of heuristic has been successful in choosing pivots [10]. Its complexity is $O(k^3 n)$.

We show experiments in Fig. 11 for uniformly distributed data in Figs. 11a and 11b, and Gaussian data in Figs. 11c and 11d, using the setup of previous sections. As can be seen, no significant improvement is obtained with the different heuristics. In some cases, random selection is even better than the alternatives. Other experiments, artificially choosing the permutants as the centers used to generate the Gaussian data, failed as well.

5 Conclusion and Future Work

We have presented a new method for probabilistic proximity searching in metric spaces. It is based on comparing the proximity ordering toward a set of distinguished objects (called permutants). We show that this ordering is a very good predictor of the relevance of points to the query. This leads to a very strong probabilistic proximity search algorithm which needs to scan just a small fraction of the database to obtain most of the relevant answers. Our technique is by far better than any other existing proposal that we are aware of.

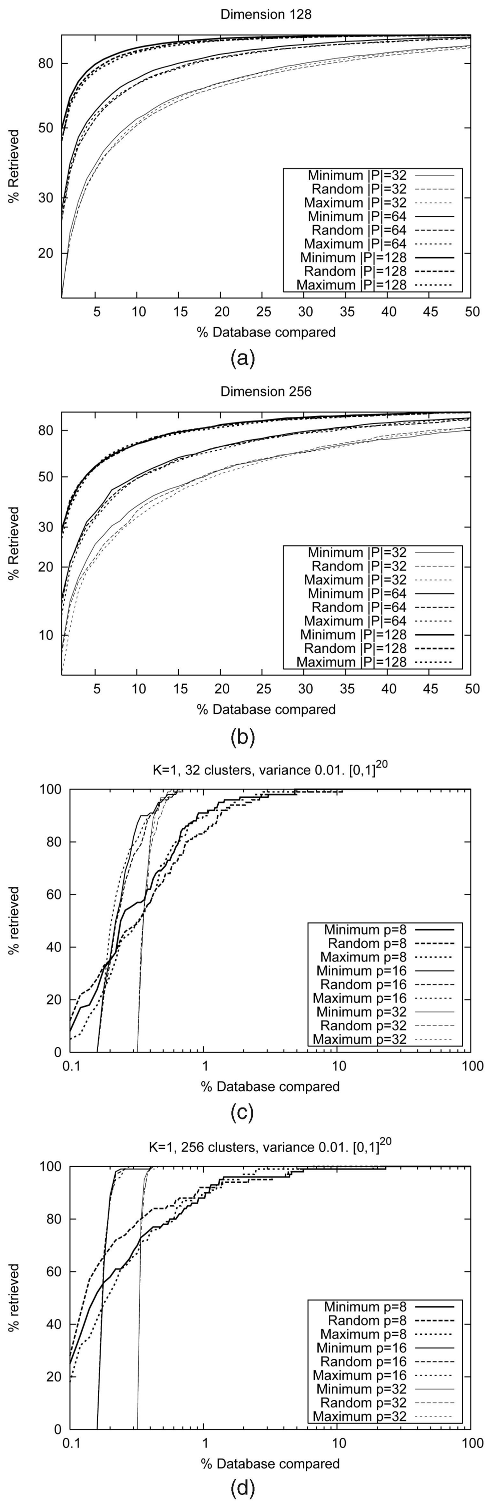


Fig. 11. Different heuristics to select permutants. In (a) and (b), range searching that retrieves 0.05 percent of the database on uniform data. In (c) and (d), 1-NN on clustered data.

Our proposal is very simple to implement and has immediate applications to many pattern recognition problems, as well as in other areas that use proximity searching and can tolerate (very good) approximations to the exact solutions to proximity queries. One application we have pursued was to use our technique as an oracle to choose the pivots in AESA, the best exact proximity search algorithm: We use S_ρ instead of L_1 [25]. The result, iAESA, achieves a relevant improvement upon an algorithm that has stood out as unbeatable for 20 years. Another idea we are pursuing is to use our algorithm to build approximate K-NN graphs, which are useful for many applications including proximity searching [34]. Our preliminary results indicate that we almost always obtain the correct K-NN graph at a very low cost compared to exact algorithms such as [35].

On the other hand, several aspects of our technique deserve more research. One challenge is to reduce CPU times. Although we have shown that permutants obtain good CPU times when the distance function is moderately expensive to compute, it might be possible to do better. In particular, our best current solutions still take time proportional to the database size (albeit with a small constant in practice). Another is to devise new methods to determine where to stop the scanning so as to achieve some expected quality in the answer. The method we proposed requires training. It may be possible to instead use the history of the updates to the answer produced by the current query to predict its future behavior.

Appendix

Due to space limitations, the Appendix is available only in electronic form at http://doi.ieeecomputersociety.org/10.1109/TPAMI.2007-70815.

Acknowledgments

The research of E. Chavez and K. Figueroa was supported by CONACyT and the research of K. Figueroa and G. Navarro was supported by the Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile. A preliminary partial version of this paper appeared in the *Proceedings of the Fourth Mexican International Conference on Artificial Intelligence*, pages 405-414, LNAI Series volume 3789.

References

[1] C. Aggarwal, "Re-Designing Distance Functions and Distance-Based Applications for High Dimensional Data," *ACM SIGMOD,* vol. 30, no. 1, pp. 13-18, 2001.

[2] C. Aggarwal, A. Hinneburg, and D. Keim, "On the Surprising Behavior of Distance Metrics in High Dimensional Spaces," *Proc. Eighth Int'l Conf. Database Theory,* pp. 420-434, 2001.

[3] A. Arslan and O. Egecioglu, "Efficient Algorithms for Normalized Edit Distance," *J. Discrete Algorithms,* vol. 1, no. 1, pp. 3-20, 2000.

[4] S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu, "An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimension," *Proc. Fifth Symp. Discrete Algorithms,* pp. 573-583, 1994.

[5] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu, "Proximity Matching Using Fixed-Queries Trees," *Proc. Fifth Combinatorial Pattern Matching,* pp. 198-212, 1994.

[6] R. Baeza-Yates and B. Ribeiro, *Modern Information Retrieval.* Addison-Wesley, 1999.

[7] J. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Comm. ACM,* vol. 18, no. 9, pp. 509-517, 1975.

[8] S. Brin, "Near Neighbor Search in Large Metric Spaces," *Proc. 21st Very Large Databases,* pp. 574-584, 1995.

[9] W. Burkhard and R. Keller, "Some Approaches to Best-Match File Searching," *Comm. ACM,* vol. 16, no. 4, pp. 230-236, 1973.

[10] B. Bustos and G. Navarro, "Probabilistic Proximity Search Algorithms Based on Compact Partitions," *J. Discrete Algorithms,* vol. 2, no. 1, pp. 115-134, 2003.

[11] E. Chávez and K. Figueroa, "Faster Proximity Searching in Metric Data," *Proc. Mexican Int'l Conf. Artificial Intelligence,* pp. 222-231, 2004.

[12] E. Chávez, J. Marroquín, and R. Baeza-Yates, "Spaghettis: An Array Based Algorithm for Similarity Queries in Metric Spaces," *Proc. Sixth String Processing and Information Retrieval,* 1999.

[13] E. Chávez, J.L. Marroquin, and G. Navarro, "Fixed Queries Array: A Fast and Economical Data Structure for Proximity Searching," *Multimedia Tools and Applications,* vol. 14, no. 2, pp. 113-135, 2001.

[14] E. Chávez and G. Navarro, "Probabilistic Proximity Search: Fighting the Curse of Dimensionality in Metric Spaces," *Information Processing Letters,* vol. 85, no. 1, pp. 39-46, 2003.

[15] E. Chávez and G. Navarro, "A Compact Space Decomposition for Effective Metric Indexing," *Pattern Recognition Letters,* vol. 26, no. 9, pp. 1363-1376, 2005.

[16] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín, "Proximity Searching in Metric Spaces," *ACM Computing Surveys,* vol. 33, no. 3, pp. 273-321, 2001.

[17] P. Ciaccia and M. Patella, "PAC Nearest Neighbor Queries: Approximate and Controlled Search in High-Dimensional and Metric Spaces," *Proc. 16th Int'l Conf. Data Eng.,* pp. 244-255, 2000.

[18] P. Ciaccia and M. Patella, "Searching in Metric Spaces with User-Defined and Approximate Distances," *ACM Trans. Database Systems,* vol. 27, no. 4, pp. 398-437, 2002.

[19] P. Ciaccia, M. Patella, and P. Zezula, "M-Tree: An Efficient Access Method for Similarity Search in Metric Spaces," *Proc. 23rd Conf. Very Large Databases,* pp. 426-435, 1997.

[20] K. Clarkson, "Nearest Neighbor Queries in Metric Spaces," *Discrete Computational Geometry,* vol. 22, no. 1, pp. 63-93, 1999.

[21] K. Doheryu, R. Adams, and N. Davey, "Non-Euclidean Norms and Data Normalisation," *Proc. 12th European Symp. Artificial Neural Networks,* 2004.

[22] R. Duda, P. Hart, and D. Stork, *Pattern Classification,* second ed. John Wiley & Sons, 1973.

[23] O. Egecioglu, "Parametric Approximation Algorithms for High-Dimensional Euclidean Similarity," *Proc. Conf. Principles and Practice of Knowledge Discovery in Databases,* pp. 79-90, 2001.

[24] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing Top k Lists," *SIAM J. Discrete Math.,* vol. 17, no. 1, pp. 134-160, 2003.

[25] K. Figueroa, E. Chávez, G. Navarro, and R. Paredes, "On the Least Cost for Proximity Searching in Metric Spaces," *Proc. Fifth Workshop Efficient and Experimental Algorithms,* pp. 279-290, 2006.

[26] D. Harman, "Overview of the Third Text REtrieval Conf.," *Proc. Third Text REtrieval Conf.,* NIST Special Publication 500-207, pp. 1-19, 1995.

[27] G. Hjaltason and H. Samet, "Index-Driven Similarity Search in Metric Spaces," *ACM Trans. Database Systems,* vol. 28, no. 4, pp. 517-580, 2003.

[28] P. Howarth and S. Rüger, "Fractional Distance Measures for Content-Based Image Retrieval," *Proc. 27th European Conf. IR Research,* pp. 447-456, 2005.

[29] G.T. Toussaint, "Computational Geometric Problems in Pattern Recognition," *Pattern Recognition Theory and Applications,* O.J. Kittler, ed., NATO ASI series, 1981.

[30] I. Kalantari and G. McDonald, "A Data Structure and an Algorithm for the Nearest Point Problem," *IEEE Trans. Software Eng.,* vol. 9, no. 5, 1983.

[31] A. Marzal and E. Vidal, "Computation of Normalized Edit Distance and Applications," *IEEE Trans. Pattern Analysis and Machine Intelligence,* vol. 15, no. 9, pp. 926-932, Sept. 1993.

[32] L. Micó, J. Oncina, and E. Vidal, "A New Version of the Nearest-Neighbor Approximating and Eliminating Search (AESA) with Linear Preprocessing-Time and Memory Requirements," *Pattern Recognition Letters,* vol. 15, pp. 9-17, 1994.

[33] G. Navarro, "Searching in Metric Spaces by Spatial Approximation," *Very Large Databases J.,* vol. 11, no. 1, pp. 28-46, 2002.

[34] R. Paredes and E. Chávez, "Using the k-Nearest Neighbor Graph for Proximity Searching in Metric Spaces," *Proc. 12th String Processing and Information Retrieval*, pp. 127-138, 2005.

[35] R. Paredes, E. Chávez, K. Figueroa, and G. Navarro, "Practical Construction of k-Nearest Neighbor Graphs in Metric Spaces," *Proc. Fifth Workshop Efficient and Experimental Algorithms,* pp. 85-97, 2006.

[36] R. Paredes and G. Navarro, "Optimal Incremental Sorting," *Proc. Eighth Workshop Algorithm Eng. and Experiments,* pp. 171-182, 2006.

[37] P. Phillips, H. Wechsler, J. Huang, and P. Rauss, "The FERET Database and Evaluation Procedure for Face Recognition Algorithms," *Image and Vision Computing J.,* vol. 16, no. 5, pp. 295-306, 1998.

[38] H. Samet, *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann, 2005.

[39] T. Skopal, "On Fast Non-Metric Similarity Search by Metric Access Methods," *Proc. Int'l Conf. Extending Database Technology,* pp. 718-736, 2006.

[40] E. Vidal, "An Algorithm for Finding Nearest Neighbors in (Approximately) Constant Average Time," *Pattern Recognition Letters,* vol. 4, pp. 145-157, 1986.

[41] D. White and R. Jain, "Algorithms and Strategies for Similarity Retrieval," Technical Report VCL-96-101, Visual Computing Laboratory, Univ. of California San Diego, July 1996.

[42] P. Yianilos, "Excluded Middle Vantage Point Forests for Nearest Neighbor Search," DIMACS Implementation Challenge, *Proc. Int'l Workshop Algorithm Eng. and Experimentation,* 1999.

[43] P.N. Yianilos, "Locally Lifting the Curse of Dimensionality for Nearest Neighbor Search," technical report, NEC Research Inst., June 1999.

[44] P. Zezula, G. Amato, V. Dohnal, and M. Batko, "Similarity Search: The Metric Space Approach," *Advances in Database Systems,* vol. 32, 2006.

Edgar Chavez received the MS degree in computer science from the Universidad Nacional Autonoma de Mexico and the PhD degree in computer science from the Centro de Investigacion en Matematicas, Mexico. He is currently a full professor at the Universidad Michoacana. His research interests include pattern recognition, algorithms, and information retrieval. He has published more than 50 technical contributions in international conference proceedings, journals, and book chapters. He has been the general chair of SPIRE '99, ENC '04, and LATIN '01 and the cochair of the technical program of CPM '03, AWIC '04, and AdHocNow! '05. He is a fellow of the Mexican *Sistema Nacional de Investigadores* (SNI), has been the president of the Mexican Computer Science Society, and is a member of the IEEE Computer Society.

Karina Figueroa received the MSc degree in computer science from the Universidad Michoacana de San Nicolas de Hidalgo, Mexico, and the PhD degree in computer science from the Universidad de Chile, Chile. She is currently a lecturer at the Universidad Michoacana, Mexico. Her research interests include algorithms, similarity searching, information retrieval, and pattern recognition. She has published more than 10 technical contributions in national and international conference proceedings and journals. She is a member of the IEEE Computer Society.

Gonzalo Navarro received the PhD degree in computer science from the University of Chile in 1998, where he is currently an associate professor and the head of the Department of Computer Science. He is also the head of the Millennium Nucleus Center for Web Research, the largest Chilean project in computer science research. His research interests include similarity searching, text databases, compression, and algorithms and data structures in general. He is a coauthor of a book on string matching and around 200 international papers. He has (co)chaired international conferences SPIRE '01, SCCC '04, SPIRE '05, SIGIR Posters '05, IFIP TCS '06, and ENC '07 Scalable Pattern Recognition track and he belongs to the editorial board of the *Information Retrieval Journal*. He is a member of the IEEE Computer Society.