Proximity Searching in High Dimensional Spaces with a Proximity Preserving Order ⋆

E. Chávez[1], K. Figueroa[1,2], and G. Navarro[2]

[1] Facultad de Ciencias Físico-Matemáticas, Universidad Michoacana, México.
{elchavez,karina}@fismat.umich.mx
[2] Center for Web Research, Dept. of Computer Science, University of Chile.
gnavarro@dcc.uchile.cl

Abstract. Kernel based methods (such as k-nearest neighbors classifiers) for AI tasks translate the classification problem into a proximity search problem, in a space that is usually very high dimensional. Unfortunately, no proximity search algorithm does well in high dimensions. An alternative to overcome this problem is the use of approximate and probabilistic algorithms, which trade time for accuracy.
In this paper we present a new probabilistic proximity search algorithm. Its main idea is to order a set of samples based on their distance to each element. It turns out that the closeness between the order produced by an element and that produced by the query is an excellent predictor of the relevance of the element to answer the query.
The performance of our method is unparalleled. For example, for a *full* 128-dimensional dataset, it is enough to review 10% of the database to obtain 90% of the answers, and to review less than 1% to get 80% of the correct answers. The result is more impressive if we realize that a full 128-dimensional dataset may span thousands of dimensions of clustered data. Furthermore, the concept of proximity preserving order opens a totally new approach for both exact and approximated proximity searching.

1 Introduction

Kernel methods (for example k-nearest neighbors classifiers) are widely used in many AI tasks, such as recommendation systems, distance based clustering, query by content in multimedia repositories, pattern recognition, and so on. The classifying process can be abstracted as a search problem in a general metric space. A newly arrived (unknown) object is classified according to its distances to known object samples.

Kernel based methods are appealing because they are simple to design and understand. The designer only needs to define a long list of features for each object, representing the object as a high dimensional vector. The problem then

⋆ Supported by CYTED VII.19 RIBIDI Project (all authors), CONACyT grant 36911A (first and second author) and Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile (second and third author).

translates into satisfying proximity queries in high dimensional spaces. Unfortunately, current methods for proximity searching suffer from the so-called *curse of dimensionality*. In short, an efficient method for proximity searching in low dimensions becomes painfully slow in high dimensions.

The concept of high dimensionality exists on general metric spaces (where there might be no coordinates) as well. When the histogram of distances among objects has a large mean and a small variance, the performance of any proximity search algorithm deteriorates just as when searching in high-dimensional vector spaces. In sufficiently high dimensions, no proximity search algorithm can avoid comparing the query against all the database.

Such a linear complexity solution for proximity searching does not scale because the distance function is computationally expensive. To cope with the curse of dimensionality, different relaxations on the query precision have been proposed in order to obtain a computationally feasible solution. This is called *approximate similarity searching*, as opposed to the classic *exact similarity searching*. Approximate similarity searching is reasonable in many applications because the metric space modelization already involves an approximation to reality and therefore a second approximation at search time is usually acceptable.

These relaxations use, in addition to the query, a precision parameter ε to control how far away (in some sense) we want the outcome of the query from the correct result. A reasonable behavior for this type of algorithms is to asymptotically approach to the correct answer as ε goes to zero, and complementarily to speed up the algorithm, losing precision, as ε moves in the opposite direction.

There are basically two alternatives for approximate similarity searching. A first one uses ε as a distance relaxation parameter, for example they ensure 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 [16, 8]. A second alternative takes ε in a probabilistic sense, for example ensuring that the answer of the algorithm is correct with probability at least $1-\varepsilon$. This corresponds to "probabilistic" algorithms in the usual algorithmic sense. A generic method to convert an exact algorithm into probabilistic is studied in [6, 3].

In this paper we present a new probabilistic proximity search algorithm. Its main idea is to order a set of samples based on their distance to each element. It turns out that the closeness between the order produced by an element and that produced by the query is an excellent predictor of the relevance of the element to answer the query. This is a completely new concept not previously explored in proximity searching. Interestingly enough, although such orders do permit exact proximity searching as well, they work much better for approximate proximity searching.

The performance of our method is unparalleled, even considering the great success of previous probabilistic methods. For example, for a *full* 128-dimensional dataset (that is, elements are uniformly distributed in 128 dimensions), it is enough to review 10% of the database to obtain 90% of the answers, and to review less than 1% to get 80% of the correct answers. The result is more impressive if we realize that a full 128-dimensional dataset usually corresponds to thousands of dimensions in clustered data.

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. This distance satisfies the three axioms that make the set a *metric space*: strict positiveness ($d(x,y) = 0 \Leftrightarrow x = y$), symmetry ($d(x,y) = d(y,x)$) and triangle inequality ($d(x,z) \leq d(x,y) + d(y,z)$). This distance is considered expensive to compute (think, for instance, in 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 quickly answer (i.e. with as few distance computations as possible) *range queries* and *nearest neighbor* queries. We are interested in this work in range queries, expressed as (q,r) (a point in $\mathbb{X}$ and a tolerance radius), which should retrieve all the database points at distance r or less from q, i.e. $\{u \in \mathbb{U} \ / \ d(u,q) \leq r\}$. On the other hand, nearest neighbor queries retrieve the K elements of $\mathbb{U}$ that are closest to q.

Most of the existing approaches to solve the search problem on metric spaces are *exact algorithms* which retrieve exactly the elements of $\mathbb{U}$ at distance r or less from q. In [8] most of those approaches are surveyed and explained in detail. Additionally in [13] the techniques to optimally search for nearest neighbor queries using progressively enhanced range queries are surveyed.

2.2 Approximate Proximity Searching

In this work we are more interested in approximate and probabilistic algorithms, which relax the condition of delivering the exact solution. This relaxation uses, in addition to the query, a *precision* parameter ε to control how far away (in some sense) we want the outcome of the query from the correct result.

Approximation algorithms are surveyed in depth in [16]. An example is [1], which proposes a data structure for real-valued vector spaces under any Minkowski metric L_s. The structure, called the BBD-tree, is inspired in *kd*-trees and can be used to find "$(1+\varepsilon)$ nearest neighbors": instead of finding u such that $d(u,q) \leq d(v,q) \quad \forall v \in \mathbb{U}$, they find u^* such that $d(u^*,q) \leq (1+\varepsilon) d(v,q) \quad \forall v \in \mathbb{U}$.

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 the cell is processed so as to obtain its nearest neighbor u. The search stops when no promising cells are found, i.e. when the radius of any ball centered at q and intersecting a nonempty cell exceeds the radius $d(q,p)/(1+\varepsilon)$. The query time is $O(\lceil 1 + 6k/\varepsilon \rceil^k k \log n)$, where k is the dimensionality of the vector space.

Probabilistic algorithms have been proposed both for vector spaces [1, 18, 16] and for general metric spaces [11, 6, 3].

In [3] they 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 order, they obtain more and more relevant answers to the query. A good database ordering is one that obtains most of the relevant answers by traversing a small fraction of the database. In other words, given a limited amount of work allowed, the algorithm finds each correct answer with some probability, and it can refine the answer incrementally if more work is allowed. Thus, the problem of finding good probabilistic search algorithms translates into finding a good ordering of the database.

Finally, there are approaches that combine approximation and probabilistic techniques, such as the PAC (probably approximately correct) method [9]. This also occurs with [6], where a general method based on stretching the triangle inequality offers both approximation and probabilistic guarantees.

2.3 Data Organization

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 [8].

Pivoting Schemes A *pivot* is a distinguished database element, whose distance to some other elements has been 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, we compute $d(p,q)$. Then, by the triangle inequality, $d(q,u) \geq |d(p,q) - d(p,u)|$, so if $|d(p,q) - d(p,u)| > r$ we know that $d(q,u) > r$, hence u will be filtered out without need of computing that distance.

The most basic scheme chooses k pivots $p_1 \ldots p_k$ and computes all the distances $d(p_i,u)$, $u \in \mathbb{U}$, into a table of kn entries. Then, at query time, all the k distances $d(p_i,q)$ are computed and every element u such that $D(q,u) = \max_{i=1\ldots k} |d(p_i,q) - d(p_i,u)| > r$ 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 the dimension 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 sort the table of kn entries to reduce extra CPU time, e.g. [5, 4].

Although they look completely different, several tree data structures are built on the same pivoting concept, e.g. [17]. 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 built on the elements they have. 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 much more pivots (albeit local and with coarse data). Finally, the trees can be traversed with sublinear extra CPU 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 are the pivots (different nodes can share pivots, which do not belong anymore 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 dimension increases. For very high dimensional data the best data structure is just a linked list (a degenerate tree) [7]. Also, little is known about how to choose the pivots.

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 good chances 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, e.g. [10]. If a group has center c and covering radius r_c then, if $d(q,c) > r + r_c$, it can be wholly discarded. The geometric shape of the above scheme corresponds to a ball around c. In high dimensions, all the balls tend to overlap and even a query with radius zero has to enter many balls.

This overlap problem can be largely alleviated with the second approach, e.g. [2, 14]. The set of centers is chosen and every database element is assigned to 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 group of c_j.

3 A New Probabilistic Search Algorithm

In this section we describe our contribution in the form of a new probabilistic algorithm based on a new indexing technique, which cannot be classified as pivot based nor compact partitioning.

We select a subset $\mathbb{P} = \{p_1, \ldots, p_k\} \subseteq \mathbb{U}$, in principle at random. Our index consists of a permutation of $\mathbb{P}$ for each element $u \in \mathbb{U}$. That is, each database element u reorders $\mathbb{P}$ according to the distances of the elements p_i to u. Intuitively, if two elements u and v are close to each other, their two permutations should be similar. In particular, if $d(u,v) = 0$, the two permutations must coincide.

At query time we compute the same permutation of $\mathbb{P}$ with respect to the query q. Then we order the database according to how similar are the permutations of the elements $u \in \mathbb{U}$ to the permutation of q. We expect that elements with orders more similar to that of q will also be spatially closer to q.

We now precise and formalize the ideas.

3.1 Index Process

Each element $u \in \mathbb{X}$ defines a *preorder* $\leq_u$ in $\mathbb{P}$. For every pair $y, z \in \mathbb{P}$ we define:

$$y \leq_u z \quad \Leftrightarrow \quad d(u,y) \leq d(u,z).$$

The relation $\leq_u$ is a preorder and not an order because two elements can be at the same distance to u, and thus $\exists y \neq z$ such that $y \leq_u z \ \wedge \ z \leq_u y$.

This preorder is not sufficient to derive a permutation in $\mathbb{P}$. Fortunately, preorder $\leq_u$ induces a total order in the quotient set. Let us call $=_u$ the equivalence related to preorder $\leq_u$, that is, $y =_u z$ iff $d(u,y) = d(z,u)$, and let us also define $y <_u z$ as $y \leq_u z \ \wedge \ y \neq_u z$.

We associate each preorder $\leq_u$ a permutation Π_u of $\mathbb{P}$ as follows:

$$\begin{aligned} \Pi_u \quad &= \quad (p_{i_1},\ p_{i_2},\ \ldots,\ p_{i_k}) \\ &\quad \text{such that } \forall 1 \leq j < k,\ (p_{i_j} <_u p_{i_{j+1}}) \vee (p_{i_j} =_u p_{i_{j+1}} \ \wedge \ i_j < i_{j+1}). \end{aligned}$$

That is, elements in Π_u are essentially sorted by $\leq_u$, and we break ties using the identifiers in $\mathbb{P}$ (any consistent way to break ties would do).

Note that our index needs $nk \log k$ bits, while pivot-based index using k pivots require typically $nk \log n$ bits.

3.2 Search Process

Let (q, r) be a query with positive search radius $r > 0$. At query time we compute the permutation Π_q. Once we have the permutation induced by the query, we will sort $\mathbb{U}$ according to how much does each Π_u differ from Π_q. Then we will traverse the database in this order, hoping to find the closest elements to u soon.

To this end we use Spearman's Footrule [12] as our similarity measure between permutations, denoted $F(\Pi_q, \Pi_u)$. We sum the differences in the relative position of each element of the permutations. That is, for each $p_i \in \mathbb{P}$ we compute its position in Π_u and Π_q. We interpert $\Pi_u(p_i)$ as the position of element p_i in Π_u. Then we sum up the absolute values of the differences $\Pi_u(p_i) - \Pi_q(p_i)|$. Formally,

$$F(\Pi_q, \Pi_u) = \sum_{1 \leq i \leq k} |\Pi_u(p_i) - \Pi_q(p_i)|$$

Thus, we traverse elements $u \in \mathbb{U}$ from smallest to largest $F(\Pi_q, \Pi_u)$.

Let us give an example of $F(\Pi_q, \Pi_u)$. Let $\Pi_q = p_1, p_2, p_3, p_4, p_5, p_6$ be the permutation of the query, and $\Pi_u = p_3, p_6, p_2, p_1, p_5, p_4$ be the permutation of an element $u \in \mathbb{U}$. 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: $3-1, 6-2, 3-2, 4-1, 5-5, 6-4$, and the sum of all these differences is $F(\Pi_q, \Pi_u) = 12$.

There are other similarity measures between permutations [12], like: Spearman's Rho and Kendall's Tau. On section 4 we will show that both have the same performance as Spearman's Footrule for our algorithm.

4 Experimental Validation

We made some experiments using uniformly distributed sets in the unitary cube. We used full dimensional dataset of 128, 256, 512 and 1024 dimensions. A full 128-dimensional dataset may span thousands of dimensions in clustered data.

The database size was 10,000 objects and the query retrieved 0.05% of database. Another parameter of our algorithm is the size of permutation. In this set of experiment we use 128 and 256 elements for $\mathbb{P}$. We compare our technique versus a classical technique of pivots using the same amount of pivots, even though this represent four more times the memory that we use in our algorithm. In other words, if we restricted the two algorithms to use the same memory, the results would be even better.

For the pivots algorithm, we calculated for each element u in the database its distance $L_\infty = \max_{p \in \mathbb{P}} |d(q,p) - d(p,u)|$. The elements in the database were sorted by this value and compared against the query taking them in this order, as in previous work [3].

In Figure 1 we show the performance of our technique. We used permutations of 128 elements and 256 for the left and right plots respectively. The x axis represents the size of the database examined, the y axis is the percentage of the answer retrieved (or the probability of returning a given answer element). Lines with the word *piv* refer to the classical pivot algorithm.

Retrieving 90% of the answer is good enough for most proximity searching applications. We observe that, as the dimension grows, a larger fraction of the database must be examined to obtain the same precision. This observation is true for the pivot based algorithm as well as for our new algorithm.

With pivots, in dimension 128, to retrieve 90% of the query we must examine 90% of the database. This is as bad as a random traversal. For the new algorithm, however we just fetch 5% of database to retrieve the same 90% of the results.

It can also be seen how the performance improves when we use larger k.

We compared other similarity measures between permutations, like Spearman's Rho and Kendall's Tau [12]. In Figure 2 we compare their relative performance. It can be seen that our algorithm has the same performance for all of them.

The AESA algorithm [15] uses the L_1 Minkowski metric as an oracle to select next-best candidates for pruning the database. We tested the L_1 distance to sort the database for a probabilistic algorithm based on pivots. That is, we repeated the same experiment above, changing the L_∞ distance by L_1 for the pivot-based algorithm. The results were similar to those using L_∞. In Figure 3 we can observe a mixed result in the filtration power. In the first part (scanning less than 20% of the database) the L_1 distance retrieves a larger percent of the database compared to L_∞, for dimension 128. Yet, once the above point is reached, the result is reversed. The same behavior is observed in all the dimensions considered. We emphasize that, anyway, the results are very far from what we obtain with our new technique.

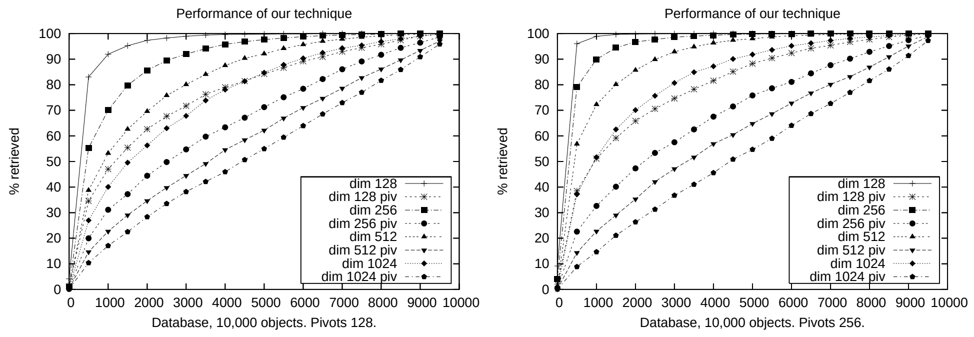

Fig. 1. Performance of the probabilistic algorithms in high dimensions. In the left side we use 128 pivots, and 256 in the right side. With our algorithm, it is possible to retrieve at least 90% of the relevant elements for the query, comparing less than 10% of the database.

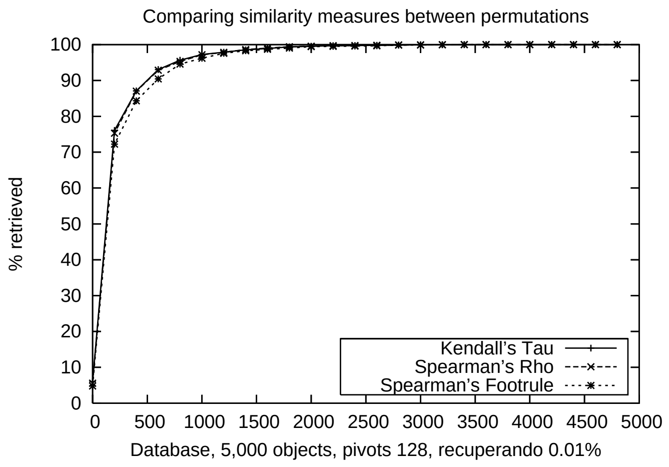

Fig. 2. Using different similarity measures between permutations. *Spearman's Footrule, Spearman's Rho* and *Kendall's Tau*. All of them have the same performance for our algorithm.

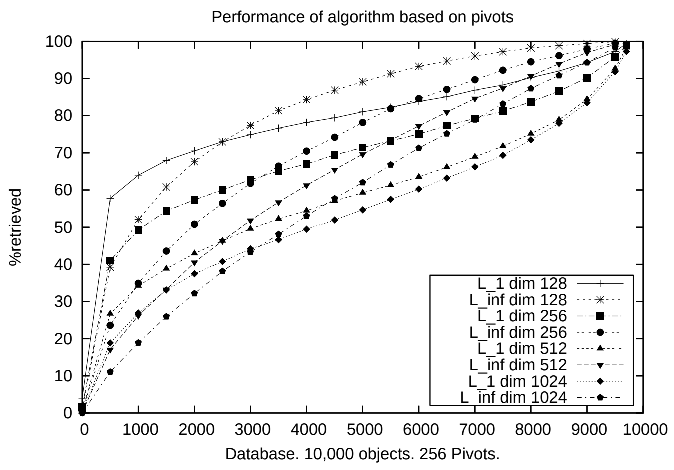

Fig. 3. Comparison between L_1 and L_∞ Minkowski metrics to sort the database on pivot-based algorithm, using 256 pivots and retrieving 0.05% of the database.

5 Conclusion and Future Work

In this paper we introduced a new probabilistic algorithm based on a proximity preserving order for metric range queries. We demonstrated experimentally that the proximity preserving order is an excellent oracle for selecting the best candidates. This has applications in both approximate and exact proximity searching algorithms.

We show experimentally that, even in very high dimensions, it is possible to retrieve 90% of the correct answer for the query, comparing less than 3% of the database, using just 256 bytes per database element.

AESA [15] is the best exact algorithm for proximity searching. It is seldom used because of its $O(n^2)$ memory requirements (as it requires saving the $O(n^2)$ distances between database elements), which do not scale for large databases. A rationale for the success of AESA is the use of the L_1 distance as an oracle. Since in our experiments the L_1 ordering is beaten by Spearman's Footrule, it would be very interesting to implement AESA using our proximity preserving order as the oracle for selecting the best next candidate for database pruning. We are currently pursuing this line with promising results.

Although our algorithm saves many distance computations, it still uses a linear number of *side CPU computations*. While this is fairly good in many applications because the distance computations are the leading complexity operations, we are working on a data structure that permits sublinear side computations as well.

References

1. S. Arya, D. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimension. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*, pages 573–583, 1994.
2. S. Brin. Near neighbor search in large metric spaces. In *Proc. 21st Conference on Very Large Databases (VLDB'95)*, pages 574–584, 1995.
3. B. Bustos and G. Navarro. Probabilistic proximity search algorithms based on compact partitions. *Journal of Discrete Algorithms (JDA)*, 2(1):115–134, 2003.
4. E. Chávez and K. Figueroa. Faster proximity searching in metric data. In *Proc. of the Mexican International Conference in Artificial Intelligence (MICAI)*, Lecture Notes in Computer Science, pages 222–231. Springer, 2004.
5. E. Chávez, J.L. Marroquin, and G. Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications (MTAP)*, 14(2):113–135, 2001.
6. E. Chávez and G. Navarro. Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Inf. Process. Lett.*, 85(1):39–46, 2003.
7. E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.
8. E. Chávez, G. Navarro, R. Baeza-Yates, and J.L. Marroquin. Proximity searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
9. P. Ciaccia and M. Patella. Pac nearest neighbor queries: Approximate and controlled search in high-dimensional and metric spaces. In *Proc. 16th Intl. Conf. on Data Engineering (ICDE'00)*, pages 244–255. IEEE Computer Society, 2000.
10. P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of the 23rd Conference on Very Large Databases (VLDB'97)*, pages 426–435, 1997.
11. K. Clarkson. Nearest neighbor queries in metric spaces. *Discrete Computational Geometry*, 22(1):63–93, 1999.
12. R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. *SIAM J. Discrete Math.*, 17(1):134–160, 2003.
13. G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. Database Syst.*, 28(4):517–580, 2003.
14. G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal (VLDBJ)*, 11(1):28–46, 2002.
15. E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.
16. D. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, La Jolla, California, July 1996.
17. P. Yianilos. Excluded middle vantage point forests for nearest neighbor search. In *DIMACS Implementation Challenge, ALENEX'99*, Baltimore, MD, 1999.
18. P. N. Yianilos. Locally lifting the curse of dimensionality for nearest neighbor search. Technical report, NEC Research Institute, Princeton, NJ, June 1999.