Improving the Space Cost of k-NN Search in Metric Spaces by Using Distance Estimators

Benjamin Bustos and Gonzalo Navarro *

Abstract

Similarity searching in metric spaces has a vast number of applications in several fields like multimedia databases, text retrieval, computational biology, and pattern recognition. In this context, one of the most important similarity queries is the k nearest neighbor (k-NN) search. The standard best-first k-NN algorithm uses a lower bound on the distance to prune objects during the search. Although optimal in several aspects, the disadvantage of this method is that its space requirements for the priority queue that stores unprocessed clusters can be linear in the database size. Most of the optimizations used in spatial access methods (for example, pruning using MinMaxDist) cannot be applied in metric spaces, due to the lack of geometric properties. We propose a new k-NN algorithm that uses *distance estimators*, aiming to reduce the storage requirements of the search algorithm. The method stays optimal, yet it can significantly prune the priority queue without altering the output of the query. Experimental results with synthetic and real datasets confirm the reduction in storage space of our proposed algorithm, showing savings of up to 80% of the original space requirement.

Keywords: similarity search, metric spaces, k-NN search.

1 Introduction

The concept of similarity search has applications in a vast number of fields. For example, content-based retrieval of similar objects in multimedia databases can be very useful for industrial applications, medicine, molecular biology,

*Authors' affiliation: Center for Web Research, Department of Computer Science, University of Chile. E-mail: {bebustos,gnavarro}@dcc.uchile.cl. Funded by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile (both authors), Agencia Española de Cooperación Internacional AECI A/8065/07 (both authors), and FONDECYT Projects 11070037 (first author) and 1-080019 (second author).

among others. In the case of industrial applications, the engineering and industrial design, the animation, and the entertainment industry heavily rely on digitized models of products or parts thereof. Given effective retrieval capabilities, the re-use of content from existing repositories can support a more efficient production processes [8]. In medical imaging applications, often 2D and 3D volume data are generated, e.g., using MRI scans. A possible application lies in automatic diagnosis support by analysis of organ deformations, via matching the actual images with medical databases of known deformations [15]. In molecular biology, structural classification is a basic task. This classification can be supported by geometric similarity search, where proteins and molecules are modeled as 3D objects, which can be compared against bio-molecular reference databases using similarity measures that consider geometry, electric properties, and others [1].

Other applications for similarity search include machine learning and classification (where a new element must be classified according to its closest existing element); image quantization and compression (where only some vectors can be represented and those that cannot must be coded as their closest representable point); text retrieval (where we look for words in a text database allowing a small number of errors, or we look for documents which are similar to a given query or document); sequence comparsion in computational biology (where we want to find a DNA or protein sequence in a database allowing some errors due to typical variations); etc.

All those applications have some common characteristics [5]. There is a universe of *objects*, and a nonnegative *distance function* defined among them, which in many interesting cases satisfies the triangle inequality. The smaller the distance between two objects, the more similar they are. This distance is assumed to be expensive to compute. We have a finite *database*, which is a subset of the universe of objects and can be preprocessed (to build an index, for instance). Later, given a new object from the universe, we must retrieve all similar elements found in the database.

A particular case of this problem arises when the space is $\mathbb{R}^d$ (a *vector space*). There are effective methods for this case, such as the kd-tree, R-tree, and X-tree, among others [2]. However, there are many applications where the space cannot be regarded as d-dimensional, for example in string similarity problems that appear in text retrieval or computational biology applications. We focus in this paper in general metric spaces, although the solutions are well suited also for d-dimensional spaces. Moreover, when d is large, methods tailored for vector spaces tend to fail and the metric space approach might be more successful.

We focus on a popular type of query called k-NN query, which aims at

finding the k database objects closest to a given query. The best known solutions to this problem (in terms of distance computations) have a serious memory problem, as they might require to maintain in memory a priority queue of database objects. This priority queue can be as large as the database itself. We propose an improved version of these k-NN search algorithms, which uses *distance estimators* to reduce their storage requirements without altering the number of distance evaluations nor the outcome of the algorithm.

We present experimental results on various synthetic and real datasets. The experiments confirm the reduction in storage space necessary to run our proposed algorithm compared to the original one. The specific numbers depend on the database and the query, and can be as low as 8% in some cases. However, in other cases we show savings of more than 80%, that is, our algorithm needs less than 20% of the space required to run the original algorithm.

The paper is organized as follows. Section 2 introduces the basic concepts of efficient similarity search in metric spaces. Section 3 depicts the proposed algorithm and analyzes its cost. An experimental evaluation using well known metric indices is presented in Section 4. Finally, in Section 5 we present our conclusions and outline the future work.

2 Searching in Metric Spaces

There are excellent books [20, 23] and surveys [9, 5, 2, 14] on efficient similarity queries in metric and multidimensional databases. In this chapter, we will briefly introduce the main indexing techniques for similarity search and will motivate the space cost problem of the optimal search algorithm.

Let $\mathbb{X}$ be the universe of valid objects, and $\delta : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$ a distance function in $\mathbb{X}$. If δ satisfies the properties of a metric, that is, *strict positiveness* ($\delta(x, y) \geq 0$ and $\delta(x, y) = 0 \Leftrightarrow x = y$), *symmetry* ($\delta(x, y) = \delta(y, x)$), and the *triangle inequality* ($\delta(x, z) \leq \delta(x, y) + \delta(y, z)$), then the pair $(\mathbb{X}, \delta)$ is a *metric space*. Let $\mathbb{U} \subseteq \mathbb{X}$ be our finite database, with $|\mathbb{U}| = n$. There are two typical similarity queries in metric spaces:

- *Range query.* A range query (q, r), $q \in \mathbb{X}$, $r \in \mathbb{R}^+$, reports all database objects that are within distance r to q, that is, $\{u \in \mathbb{U},\ \delta(u, q) \leq r\}$.
- *k nearest neighbors query (k-NN).* Reports the k objects from $\mathbb{U}$ that are closest to q, that is, a set $\mathbb{C} \subseteq \mathbb{U}$ such that $|\mathbb{C}| = k$ and $\forall x \in \mathbb{C}, y \in \mathbb{U} - \mathbb{C}, \delta(x, q) \leq \delta(y, q)$.

Indexes for metric spaces can be classified into two main categories [5]: *based on pivots* and *based on compact partitions*.

Pivot-based indexes [5, 21, 7] select a number of "pivot" objects from the database, and classify all the other objects according to their distance to the pivots. The canonical pivot-based range query algorithm is as follows: Given a range query (q, r) and a set of k pivots $\{p_1, \ldots, p_k\}, p_i \in \mathbb{U}$, by the triangle inequality it follows for any $x \in \mathbb{X}$ and $1 \leq i \leq k$ that $\delta(q, x) \geq |\delta(p_i, x) - \delta(p_i, q)|$. The objects $u \in \mathbb{U}$ of interest are those that satisfy $\delta(q, u) \leq r$, so one can exclude all the objects that satisfy $|\delta(p_i, u) - \delta(p_i, q)| > r$ for some pivot p_i (exclusion condition), without actually evaluating $\delta(q, u)$. The index consists of the kn distances $\delta(u, p_i)$ between every object of the database and every pivot. At query time it is necessary to compute the k distances between the pivots and the query q in order to apply the exclusion condition. The list of objects $\{u_1, \ldots, u_m\} \subseteq \mathbb{U}$ that do not satisfy the exclusion condition must be directly checked against the query. Several indexes resort to a tree structure to avoid considering the exclusion condition for each $u \in \mathbb{U}$ individually. Each tree node p is a pivot and its subtrees correspond to ranges of distances $\delta(u, p)$. At search time, we compute $\delta(q, p)$ and need only to enter tree branches whose ranges of distances intersect $[\delta(q, p) - r, \delta(q, p) + r]$.

Indexes based on compact partitions [5, 6, 4, 17] divide the space into *zones* as compact as possible. Each zone stores a representative point, called the *center*, and data that permit discarding the entire zone at query time without measuring the actual distance from the objects of the zone to the query. Each zone can be recursively partitioned into more zones, inducing a *search hierarchy*. There are two general criteria for partitioning the space: *Voronoi partition* and *covering radius*. The *Voronoi diagram* of a set of objects is a partition of the space into cells, each of which consisting of the objects closer to one particular center than to any other. A set of m centers is selected and the rest of the objects are assigned to the zone of their closest center. Given a range query (q, r), the distances between q and the m centers are computed. Let c be the closest center to q. Every zone of center $c_i \neq c$ which satisfies $\delta(q, c_i) > \delta(q, c) + 2r$ can be discarded, because its Voronoi area cannot intersect the query ball. On the other hand, the *covering radius* $cr(c)$ is the maximum distance between a center c and an object that belongs to its zone. Given a range query (q, r), if $\delta(q, c_i) - r > cr(c_i)$ then zone i cannot intersect the query ball and all its objects can be discarded.

In most cases, authors focus on solving range queries, as these can be regarded as being more basic than k-NN queries [5]. Given a technique to solve range queries, one can derive k-NN query solutions. The most naive

way is to try range queries with increasing radii until retrieving k objects or more. A more sophisticated method is to launch a range search with infinite radius, and reduce it on the fly as new database objects are compared and better candidates for the k-NN answer appear (that is, if we have already found k database objects with maximum distance r to q, then our search radius becomes r). This has been explored especially with tree-like indexes (the most popular category).

Unlike the case of range searching, where the tree traversal order is irrelevant, for k-NN search we wish to find close candidates as soon as possible, as this will determine how much of the tree is traversed. The most common traversal order is *depth-first* traversal. In this case, the tree traversal is recursive, and the criterion to try to find soon good k-NN candidates translates into traversing the children of the current node from most to least promising. Given a criterion to prefer one node over another, depth-first traversal does not achieve the optimal node traversing order because it is forced to be depth-first. On the other hand, the amount of memory it requires does not exceed the height of the tree index.

An optimal, *best-first* traversal ordering [22, 12] uses a global priority queue where unprocessed tree nodes are inserted, giving higher priority to the more promising ones, and the tree is traversed in the order given by the queue. Each extracted node inserts its children in the queue. If this priority is defined as a *lower bound to the distance* between q and any element in the subtree rooted at the tree node, then the search can finish as soon as the most promising node has a lower bound larger than the distance to the current k^{th} NN. This algorithm has been proved to be optimal in the number of nodes (i.e., disk page accesses) required [2]. It was also proved to be *range-optimal* [13], that is, the number of distance computations to find the k-NN answer is exactly that of a range search with distance $\delta(q, o_k)$, where o_k is the k^{th} NN. This range search would give the same answer as the k-NN search, so there is no penalty for not knowing $\delta(q, o_k)$ beforehand.

The algorithm has, however, an important problem, which may discourage its use in practice. The problem is the amount of memory required for the queue, which can store *as many elements as the database itself*. The non-optimal depth-first-search algorithm may be preferable for its much lower space consumption, proportional to the depth of the hierarchy. Samet [19, 20] proposed a technique to alleviate this problem, based on computing an upper bound to the distance between q and any subtree element, and using the fact that one knows that a subtree must have at least one element within that upper bound distance to q. We refine this idea, which will be described later as a particular case of ours. Our refinement consists in us-

Symbol	Definition
$k \in \mathbb{N}$	# of NN to be retrieved
$B \subseteq \mathbb{U}$	Ball (cluster of objects)
$B.c \in \mathbb{U}$	Center of ball B
$B.cr \in \mathbb{R}$	Covering radius of B
$B.bsize \in \mathbb{N}$	# objects inside B
$B.children$	Set of children balls of B
$B.lbound \in \mathbb{R}$	Lower bound distance from B to q
$B.ubound \in \mathbb{R}$	Upper bound distance from B to q
B_b	Bubble of B
$x.csize \in \mathbb{N}$	Size of bubble or object in C
Q	Queue with unprocessed balls
C	Queue with NN candidates
$C.maxUB \in \mathbb{R}$	Max. upper bound distance in C
$C.size \in \mathbb{N}$	Sum of all $x.csize$ in C
$x.distq \in \mathbb{R}$	Distance from x to q

Table 1: Summary of symbols

ing the information on the number of elements in the ball, all of which lie within that upper bound. Unlike Samet's approach, ours translates into a relevant contribution even on Euclidean spaces, where Samet's approach is never better than the MinMaxDist estimator [18].

3 Algorithm Description

Table 1 lists the notation used throughout this section. We assume that the database index is a *hierarchical data structure* which groups close objects in clusters. We will refer to these clusters in our metric space as *balls* (for their shape resemblance in Euclidean spaces, in many index structures). A ball B from the index contains a number of objects from the database, represented by $B.bsize$. The *center* of B is a distinguished object $B.c \in B$, usually selected trying to minimize the *covering radius* of B, $B.cr = \max\{\delta(B.c, b),\ b \in B\}$, that is, the maximum distance between $B.c$ and any other object in B. Those elements $b \in B$ can be recursively organized into balls, which descend from B forming a *search hierarchy*. As seen in Section 2, this type of hierarchical clustering index is very popular (for example see [6, 4, 17]). There are other indexes that, although less obviously, can be considered as belonging to this scheme (for example, pivot-based indexes organized in trees).

Given a query q and a ball B, the *lower bound distance* from q to B, $B.lbound$, is a lower bound to $\delta(q, b)$ for any $b \in B$. Similarly, the *upper*

bound distance from q to B, $B.ubound$, is an upper bound to $\delta(q, b)$ for any $b \in B$.[1] Figure 1 illustrates both bounds in 2D space using the Euclidean distance and the covering radius. On index structures for vector spaces that use minimum bounding rectangles (MBRs), the lower (upper) bound distance can be defined as the minimum (maximum) distance from q to the MBR. These distance bounds can be used to prune the search while performing a similarity query. For example, if we know that the upper bound distance to the k^{th} NN candidate at some point of the search is $maxUB$, and we know that $maxUB < B.lbound$ for a ball B, then it is not possible that an object inside B is closer to q than any of the current k-NN candidates. Thus, one can safely discard B and all its descent.

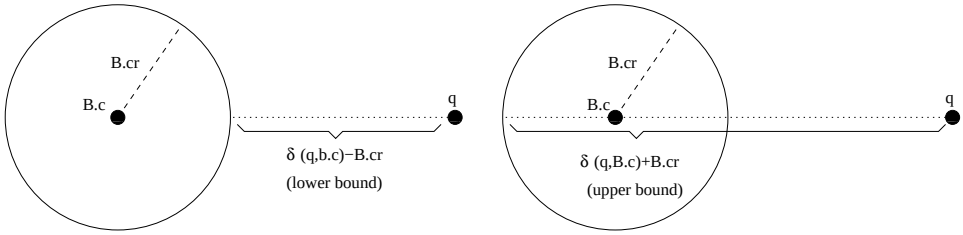

Figure 1: Distance estimators: Lower and upper bound distance from q to any object on the ball. This example shows the estimators based on the covering radius.

3.1 Standard Best-first k-NN Algorithm

The best-first k-NN search algorithm [22, 12] uses two priority queues, one (Q) that contains the balls not yet processed (also called *active page list* in the literature), and the other (C) with the k-NN candidate list. A ball is stored in Q if it is not yet processed but its parent was already processed. At each step of the search, the algorithm removes the ball B from Q with smallest $B.lbound$. The distance between the center of each children of B and q is computed, inserting in C all centers that are closer than the current k^{th} NN candidate. The children balls are inserted in Q. The algorithm ends when Q becomes empty or when the minimum $lbound$ from a ball in Q is greater than the distance to q of the current k^{th} NN candidate, as at this point no other ball can improve the current candidate list.

[1] Our proposed algorithms are general and will work with any hierarchical index structure with appropriately defined distance estimators. For simplicity, we will describe them using the covering radius for computing the distance bounds.

Note that the size of C never exceeds k, but Q can be as large as the database itself. As explained before (Section 2), this algorithm is optimal in several aspects but it has a serious memory usage problem (for Q), which our proposal seeks to alleviate.

3.2 Our Proposal

As in the basic algorithm, we use two priority queues, Q and C. Q still contains unprocessed balls whose center has already been processed, and is sorted by *lbound*. C is still sorted by *ubound*, but now it will contain a mixture of objects and *bubbles*. A *bubble* B_b in C corresponds to a ball B that exists in Q, but the bubble itself does not contain any element. From the bubble B_b we only know the upper bound $B.ubound$ and size $B.bsize$ of its corresponding ball B (that is, $B.bsize$ indicates the number of objects $u \in \mathbb{U}$ that are inside B). The existence of bubble B_b in C just tells us that there exist $B.bsize$ elements at distance at most $B.ubound$ from q, yet we still do not know those elements. With this upper bound information, we can prune irrelevant elements from Q earlier. For example, assume we find B such that $B.bsize > k$. Before knowing the elements of B, we find B' such that $B'.lbound \geq B.ubound$. At this point we can discard B', as we know that we will get enough better k-NN candidates from B, even when we have not yet obtained them.

Our algorithm maintains the following invariants. We assume that there are at least k elements in the database, otherwise the query is trivial. For simplicity we assume that balls do not directly contain objects in general, just further balls. The leaf balls of the tree contain balls that have only one element, so the balls become empty once one removes their center. This does not restrict the algorithm in any way, it is just a way to present it.

(i) We process the database hierarchy starting at the root, and never process a node without having processed its parent.

(ii) Any hierarchy ball not yet processed is in Q or descends from a ball in Q, yet the centers of balls in Q have already been processed.

(iii) Any object c in C is the center of a ball already processed, $c.csize = 1$.

(iv) Any bubble B_b in C corresponds to a ball B currently in Q, $B_b.csize = B.bsize - 1 > 0$.

(v) $C.size \geq k$ is the sum of $csize$'s of objects and bubbles in C. $C.maxUB$ is the maximum *ubound* in C, taking $c.ubound = c.distq$ for objects.

(*vi*) C contains the objects and bubbles with smallest *ubound* processed so far.

(*vii*) If we remove any element from C with *ubound* equal to $C.maxUB$, then $C.size < k$.

(*viii*) For any ball B, $B.lbound \leq B.ubound$, and the *lbound* (*ubound*) of any descendant of B is not smaller (larger) than $B.lbound$ ($B.ubound$).

(*ix*) (optional) For any ball B such that $B.bsize > 1$, $B.lbound < B.ubound$. This holds for many indexes and we can take advantage of it (as shown in the second point below).

The above invariants ensure the correctness of the following termination conditions.

- Assume $Q = \emptyset$ at some point. Then we have processed all the database objects (i, ii). Moreover, there cannot be bubbles in C (iv), so C contains just objects, of $csize = 1$ (iii). Therefore, C contains exactly k objects (v, vii), and those are the objects with smallest *distq* in the database (vi). Thus C is the correct answer to the query.

- Assume, at some point, that B has the smallest *lbound* in Q and $B.lbound > C.maxUB$. Because of condition ($viii$), $ubound \geq lbound$ for any element and *lbound* for a descendant of B can never be smaller than $B.lbound$. Thus, condition (vi) holds for all the database, not only for the elements processed so far (ii). Moreover, C cannot contain any bubble B'_b, because $B'_b.lbound \leq B'_b.ubound \leq C.maxUB < B.lbound$ for any ball B in Q, and ball B' must be in Q (iv). Thus the same arguments as before show that C is the correct answer to the query.

The second point above makes clear the correctness of the following observation, which is the key to the space reduction we achieve.

Observation 1 *If, for some B, $B.lbound > C.maxUB$ (or $B.lbound \geq C.maxUB$ if (ix) holds), then the output of the algorithm does not vary if we remove B and all its descent from Q.*

If condition (ix) holds, then, if B'_b were in C, it would hold $B'.csize > 1$ (iv), and thus $B'_b.lbound < B'_b.ubound$, and it would be sufficient that $B.lbound \geq C.maxUB$ to know that we have already the correct answer in

C. For simplicity, we assume for the rest of the paper that condition (ix) holds; it is already clear how to modify the algorithms otherwise.

We explain now how we set and maintain the invariants throughout the algorithm. We initialize Q with the only ball that roots the whole index (for simplicity, we assume there is only one such root, it is easy to insert several roots if so is the index structure). Its center and corresponding bubble are inserted in C (only the center if $k = 1$ or the bubble size is zero). This satisfies all the invariants. At each step of the algorithm, we extract the ball B from Q with smallest $B.lbound$. Recall that $B.c$ has already been processed. Now, to restore invariant (ii), we must insert in Q every child B' of B such that $B'.lbound < C.maxUB$ (otherwise, we know that the descent objects of B' can be immediately pruned from the search, by Obs. 1). Then, to restore invariants (iii, iv), we must insert into C every center $B'.c$ and bubble B'_b, as well as remove bubble B_b from C, if present. Actually, if B_b is in C, we will replace it with the centers $B'.c$ and bubbles B'_b, which add up the same $B_b.csize$. This replacement cannot affect invariants (v, vi) as the new *ubounds* are never larger than that of B_b $(viii$; note that $C.maxUB$ adjusts automatically as the maximum of the priority queue $C)$. Yet, we have to restore invariant (vii). We must remove from C the elements with largest *ubound* as long as $C.size \geq k$. We choose those with largest *ubound* so as to maintain (vi), and update $C.size$ and $C.maxUB$ to maintain (v). The remaining invariant (i) holds because we only access B' from its already processed parent B. Although not necessary for the correctness of the algorithm, we remove balls of Q that become irrelevant each time $C.maxUB$ is reduced (according to Obs. 1). This further diminishes the memory requirement for Q.

Algorithm 1 shows the pseudocode of the proposed k-NN search algorithm. Note that we enforce that *lbound* is increasing and *ubound* is decreasing as we descend in the hierarchy $(viii)$. Although this should hold, it might not occur automatically if we simply use, for example $B'.ubound = \delta(q, B'.c) + B'.cr$ for B' child of B, because the ball of B' could exceed spatially that of B, although we know that there cannot be objects of B' in the exceeded area (Figure 2 illustrates). Thus, the correct value is $B'.ubound = \min(B.ubound,\ \delta(q, B'.c) + B'.cr)$. We make the correction only if B_b is in C, otherwise $B.ubound > C.maxUB$ and the correction is irrelevant. The same holds for $B'.lbound = \max(B.lbound, B'.c.distq - B'.cr)$, which might permit pruning B' from Q earlier.

*Add*2 is a special procedure to insert into C, which in addition to the element and its *ubound* takes the *csize* of the element. *Add*2 takes care of updating $C.size$ and $C.maxUB$, and of maintaining invariants (vi, vii). Al-

Algorithm 1: Our proposed k-NN search algorithm

Input: $Index$, $q \in \mathbb{X}$, $k \in \mathbb{N}$
Output: k-NN

$Q \leftarrow \emptyset$;
$C \leftarrow \emptyset$;
$C.maxUB \leftarrow \infty$;
$C.size \leftarrow 0$;
$B \leftarrow$ root of $Index$;
$B.c.distq \leftarrow \delta(q, B.c)$;
$B.lbound \leftarrow B.c.distq - B.cr$;
$Q.Add(B,\ B.lbound)$;
$C.Add2(B.c,\ B.c.distq,\ 1)$;
$C.Add2(B,\ B.c.distq + B.cr,\ B.bsize - 1)$;
while $Q \neq \emptyset$ **do**
    $B \leftarrow Q.DequeueMin()$;
    if $B_b \in C$ **then**
        $ubound \leftarrow B_b.ubound$;
        $C.size \leftarrow C.size - B_b.csize$;
        $C.Remove(B_b)$;
    else $ubound \leftarrow \infty$;
    foreach $B' \in B.children$ **do**
        $B'.c.distq \leftarrow \delta(q, B'.c)$;
        $C.Add2(B'.c,\ B'.c.distq,\ 1)$;
        if $B'.bsize > 1$ **then**
            $B'.lbound \leftarrow \max(B.lbound, B'.c.distq - B'.cr)$;
            if $B'.lbound < C.maxUB$ **then** $Q.Add(B',\ B'.lbound)$;
            $minub \leftarrow \min\{ubound,\ B'.c.distq + B'.cr\}$;
            $C.Add2(B',\ minub,\ B'.bsize - 1)$;
        $Q.Shrink()$;
return C

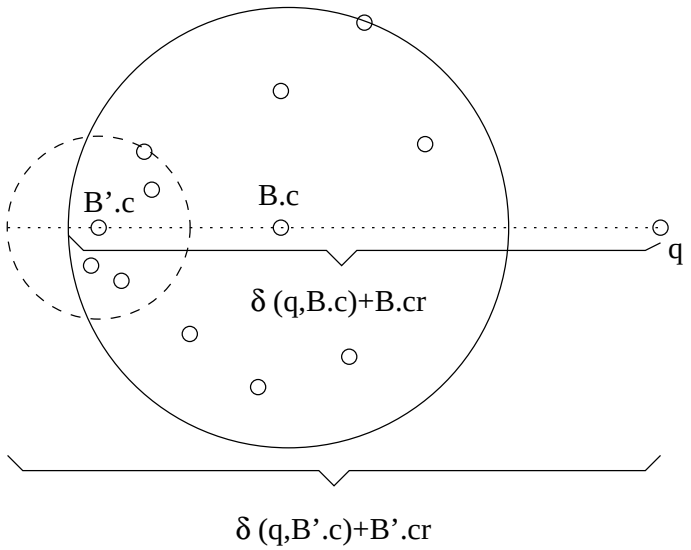

Figure 2: The correct *ubound* for B' is $\min\{B.ubound,\ \delta(q, B'.c) + B'.cr\}$. The index hierarchy ensures that no object farther than $B.cr$ from $B.c$ can be stored in B'.

gorithm 2 shows the pseudocode of the *Add*2 function for C. The algorithm assumes that C is a priority queue sorted decreasingly by *ubound*, and in case of ties, it is sorted increasingly by *csize*. This is necessary to ensure that condition (*viii*) is maintained.

Algorithm 3 shows the pseudocode for the pruning of Q, *Shrink*, called each time the maximum upper bound distance $C.maxUB$ might change, to reduce the storage requirements of the search algorithm. Note that, thanks to the use of *Shrink*, we can always finish when Q becomes empty, since if the other termination condition holds, then *Shrink* will take care of removing all the remaining elements from Q. To reduce the CPU cost associated to *Shrink*, one should call it only when $C.maxUB$ has changed (not when it might have changed, as shown for simplicity), and implement Q as a min-max heap.

A key element of our k-NN algorithm is $B.bsize$. If this value is not stored in the index, then the proposed algorithm cannot run and the best that one can do in that case is to assume $B.bsize = 2$ for internal hierarchy nodes (since $B.bsize \geq 2$, for the center and at least another point). This is precisely what was done by Samet in previous work [19], and we refine it here assuming $B.bsize$ is known. Slightly better than assuming $B.bsize = 2$ is, if B has *cb* child balls and *co* child objects, assume $B.bsize = 2 \cdot cb + co$.

Algorithm 2: *Add2* algorithm for C

Input: B, $ubound \in \mathbb{R}^+$, $csize \in \mathbb{N}$

```
1  if ubound < C.maxUB then
2      B_b ← CreateBubble(B);
3      B_b.ubound ← ubound;
4      B_b.csize ← csize;
5      C.size ← C.size + B_b.csize;
6      C.Add(B_b, B_b.ubound);
7      while C.size − C.Max().csize ≥ k do
8          C.size ← C.size − C.Max().csize;
9          C.DequeueMax();
10     if C.size ≥ k then C.maxUB ← C.max().ubound;
```

Algorithm 3: Shrink algorithm for Q

```
1  while Q ≠ ∅ and Q.Max().lbound ≥ C.MaxUB do
2      Q.DequeueMax();
```

3.3 Example

Figure 3 shows an example of a k-NN query for some $k > 1$. The index consists of a ball B which has three child balls: $B1$, $B2$, and $B3$. From the figure, it follows that $B.bsize = 1 + X + Y + Z$, and let us suppose that $X \geq k$. The figure shows the index hierarchy up to the first level. In the beginning, $B.c$ is inserted into C, as well as the bubble B_b with upper bound $B.c.distq + B.cr$. Thus, $C.size = 1 + X + Y + Z \geq k$ and if we extract the bubble with greater *ubound* then $C.size = 1 < k$, thus the invariants hold. Ball B is inserted into Q and the algorithm enters the loop. Ball B is extracted from Q, and then B_b is removed from C. Now the algorithm processes the children of B. The object $B1.c$ and the bubble $B1_b$ are inserted into C. Meanwhile, $B.c$ is removed from C because $B1.ubound < B.c.distq$ and $C.size \geq k$ with $B1_b$ and $B1.c$ in C. The maximum upper bound $C.maxUB$ is updated to $C.maxUB = B1.ubound$, $B1$ is inserted into Q, and the algorithm invokes $Q.Shrink()$. $B2$ and $B3$ will never be inserted into Q, because $C.maxUB < B2.lbound < B3.lbound$. Therefore, the maximum length of Q until this step was 1. Using the standard algorithm, $B2$ and $B3$ *will be* inserted into Q, even when they will never be removed from the queue (the algorithm will stop searching before removing them), thus

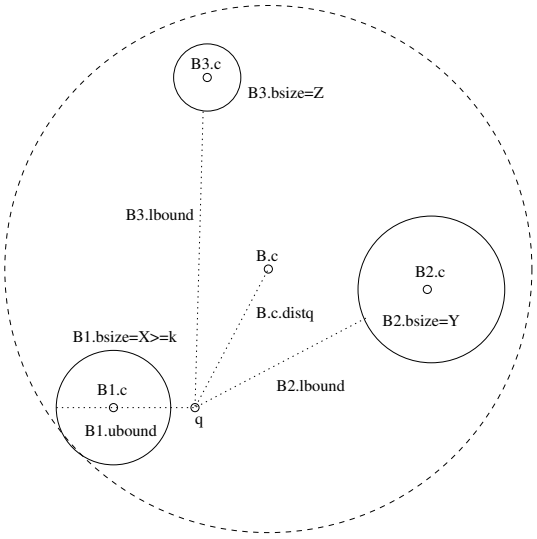

Figure 3: Example of a k-NN query. Note (visually) that all k-NN are on $B1$ and $B1.ubound < B.c.distq$.

wasting storage space.

Assume now that $B1$, $B2$, and $B3$ are inserted into Q (for example, this could be the case if the index contained several roots). The algorithm removes $B1$ from Q and processes each of its children. Then, it updates $C.maxUB$. Note that the algorithm ensures that $C.maxUB \leq B1.ubound$. Next, procedure $Q.Shrink()$ is invoked, which removes balls $B2$ and $B3$ from Q, thus diminishing the average length of Q during the search.

In both cases, the algorithm was able to prune balls $B2$ and $B3$ without processing them and at a very early stage of the search. Therefore, the storage requirements for Q was successfully diminished using our proposed algorithm.

3.4 Cost Analysis of the Proposed Algorithm

Now we compare the computational complexity of the original and our proposed k-NN search algorithms. Firstly, as both algorithms perform a best-first traversal of the index, it follows that they carry out exactly the same number of distance computations for the same query object. Thus, for this concept the CPU cost is the same on both algorithms. Moreover, this implies that our algorithm is also optimal in the number of node accesses required to answer k-NN queries, and it is range-optimal (see Section 2).

Regarding the insertion/deletions of elements in Q, the CPU cost of the original algorithm is $O(totQ \cdot \log maxQ)$, where $totQ$ is the overall number of balls ever inserted into Q and $maxQ$ the maximum size of Q across the process. The cost for our proposed algorithm is the same, noting that in our case $totQ$ and $maxQ$ will be smaller, given that we avoid some insertions into Q.

With respect to C, the CPU cost of the original algorithm is in the worst case $O(totQ \cdot \log k)$, since all centers from balls in Q may be inserted into C, and it is ensured that only one object per iteration may be extracted from C. For our proposed algorithm, in the worst case it may be possible that $totQ$ objects and $totQ$ bubbles are inserted into C, but then the algorithm may extract up to k elements from C after an insertion (cf. Algorithm 2, lines 7–9 of the pseudocode). However, it is not possible to extract more elements that those inserted into C, therefore the total CPU cost in the worst case is also $O(totQ \cdot \log k)$, that is, it is the same CPU cost compared with the original algorithm. Note that the query "$B_b \in C$" in line 13 of Algorithm 1 requires a dictionary data structure built on top of C (such as a hash table) if one wants to avoid an $O(k)$ time linear traversal. Such a table involves spending $O(k)$ extra memory, but this is usually irrelevant as k is very small in all meaningful cases.

Thus, our proposed algorithm has the same CPU cost as the original one, but it always uses less memory for Q. As previously observed, our algorithm needs to know how many objects are within each ball of the index, which also uses some memory space (one integer value per internal node, which is usually very modest). Many indexes already store this value. Otherwise, in most practical situations, there is always some free room on each index node, because it is almost impossible to completely use its assigned space (equal to the size of a disk data page), so the extra integer can be stored "for free". Also, we experimentally observed that the memory savings are at least an order of magnitude higher than the extra space used. Although not easy to guarantee analytically, the next section shows that the space savings are very significant in practice.

4 Experimental Evaluation

In this section, we show empirically that our proposed technique can achieve significant space savings. The exact amount will depend on the type of metric space and queries posed to it.

For our experiments, we used several synthetic and real-world databases:

- *Gaussian*: This is a set of synthetic databases that are formed by clusters in a vector space using different dimensionalities (8-D, 16-D, and 32-D), where the objects that conform each cluster follows a Gaussian distribution. Each Gaussian database contains 1,000 clusters, and their centers are random points with coordinates uniformly distributed in $[0, 1]$. The variance for the Gaussian distribution was set equal to 0.001 for each coordinate, to produce compact clusters. The size of each cluster is similar but not necessarily equal, and the whole dataset contains 100,000 objects. We generated 1,000 random query points, which follow the same data distribution as the database.

- *Corel Features*: The *Corel image features* contains features from 68,040 images extracted from a Corel image collection. The features are based on the color histogram (32-D), color histogram layout (32-D), co-occurrence texture (16-D), and color moments (9-D). This database is available at the *UCI KDD Archive* [10]. For our experiments, we used the color histogram (CH) and the layout histogram (LH) databases. We selected a subset of this database consisting on 65,515 images, because there were some missing features for some of the images. For each database, we choose 1,000 images at random to be used as queries.

- *Edge structure*: This database contains 20,197 feature vectors (edge structure, 18-D) extracted from the Corel image database. We selected 1,000 random objects from the database as query points.

- *English String DB*: It contains $69,069$ strings (a dictionary of English words). We selected $1,000$ random strings as query points.

We used the *Manhattan distance* as the metric for the multidimensional databases. Other metrics (e.g., Euclidean) may be used, but better results have been obtained with respect to the effectiveness of the search using the Manhattan distance [3] (which is in accordance with theoretical results [11]), and it is the (computationally speaking) cheapest Minkowski distance. For the English String DB, we used the edit distance (the minimum number of insertions, deletions and replacements performed to convert one string into another) as the metric, as it is relevant for many applications [16].

The *number of objects per cluster* was selected depending on the dimensionality of the dataset, in such a way that all objects from the cluster (plus a small header) could fit on a disk datapage. Setting the datapage size to 4 Kb, we obtained the following cluster size values: 127 (8-D), 63 (16-D), 56

(18-D), and 31 (32-D). For the English String DB, we used a cluster size of 16.

We compared the standard best-first k-NN algorithm (labeled *HS*) against ours (labeled *Ours*) and the best-first version proposed by Samet [19] (labeled *Samet*)[2]. As representative index structures, we used the *List Of Clusters* [4] and the *M-tree* [6]. The List of Clusters can be seen as a "list of balls", that is, a search hierarchy with only one level, while the M-tree is a more general hierarchical index structure. The former usually performs better on higher dimensional spaces. To compare the storage requirements of each search algorithm, we computed the mean of the maximum queue lengths ($\max\{|Q|\}$) obtained on each query, and the average length of Q over all queries. The first measure indicates how much memory (on average) needs the search algorithm to answer the k-NN query. The second measure is related with the number of disk accesses made if the queue were stored on secondary memory. All results are shown as percentages of the database size.

Figures 4 to 6 show the results obtained with the Gaussian databases. Our algorithm needs considerably less memory than the standard algorithm, especially for high dimensions. For example, our algorithm used only 18% of the memory required by the standard algorithm in 32-D and using List of Clusters ($k = 50$). In low dimensions, the gain was smaller (48% of the memory requirement of the standard algorithm), but still considerable. The List of Clusters performed better than the M-tree in terms of storage usage. In this index, our algorithm was consistently better than the simpler version by Samet. The average length of the queue was up to 5 times shorter than the standard algorithm. In the M-tree, on the other hand, all the space performances were rather similar. The charts also show that the storage efficiency degrades as k grows, especially in the case of the M-tree.

We obtained similar results with real-world datasets (see Figures 7 to 9). For example, with the color histogram database, our algorithm only used 34% of the storage requirement of the standard algorithm with List of Clusters. The average queue length was always smaller than 32% of that of the standard algorithm. Similar improvements were obtained with the layout histogram and the edge structure databases. Finally, the obtained results with the English String DB were not as good as with the other database, but the algorithm was able to save 16% of the memory used by

[2]Actually, Samet speaks mostly of depth-first algorithms [19], but the paper mentions that the best-first algorithm could be handled as well. As we are interested only in the optimal traversal order in this paper, we compare only its best-first version.

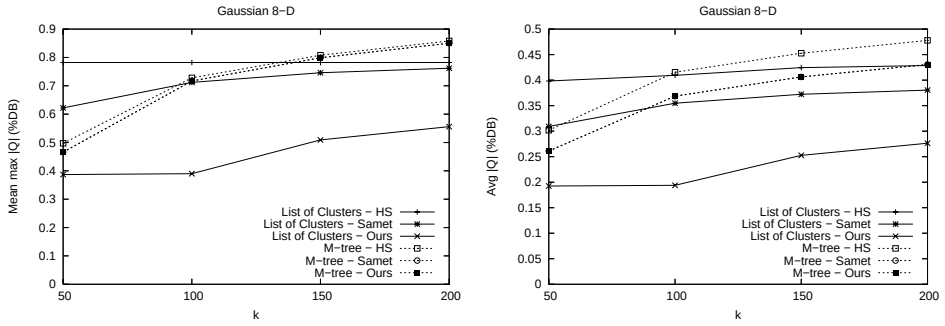

Figure 4: Gaussian 8-D: Mean of the maximum queue length (left) and average queue length (right).

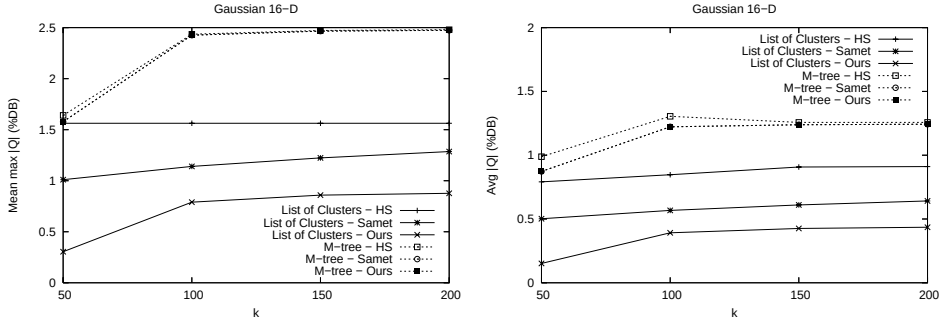

Figure 5: Gaussian 16-D: Mean of the maximum queue length (left) and average queue length (right).

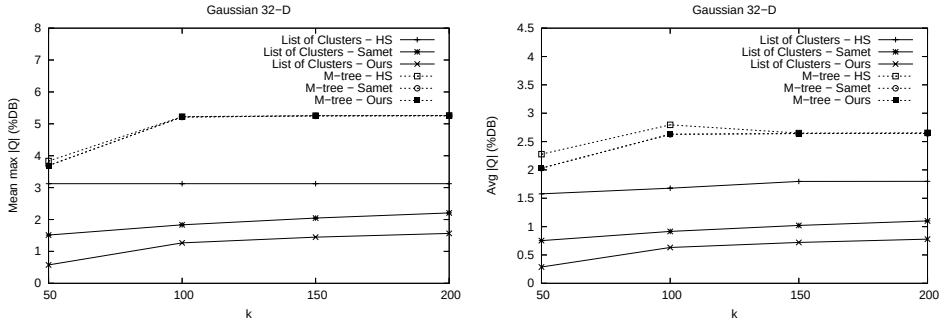

Figure 6: Gaussian 32-D: Mean of the maximum queue length (left) and average queue length (right).

the standard algorithm on average.

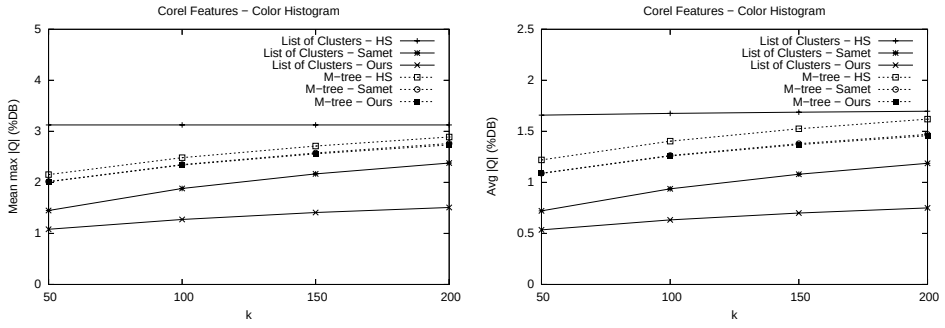

Figure 7: Corel features CH: Mean of the maximum queue length (left) and average queue length (right).

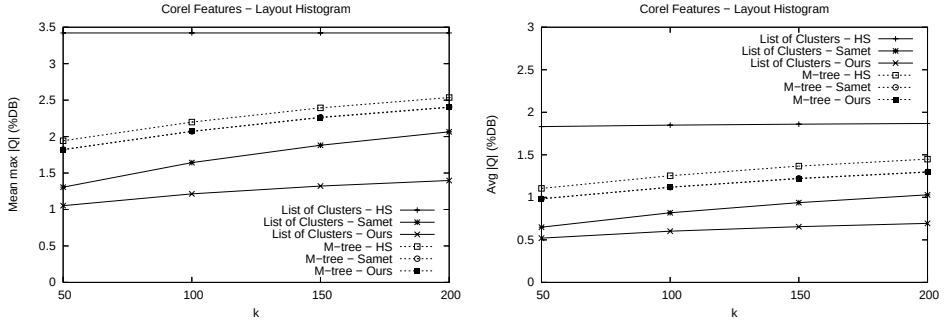

Figure 8: Corel features LH: Mean of the maximum queue length (left) and average queue length (right).

Table 2 summarizes the improvements in storage requirements of our algorithm over the standard k-NN search.

5 Conclusions

We presented an improved version of the optimal-order k-NN search algorithm, using distance estimators (such as the upper and lower bound distance to the query object) in order to reduce the storage requirements of the search algorithm. Our proposed algorithm aims to prune from the active page list, as soon as possible, all nodes from the index where it is ensured that no relevant objects can be found. We also introduce the concept of

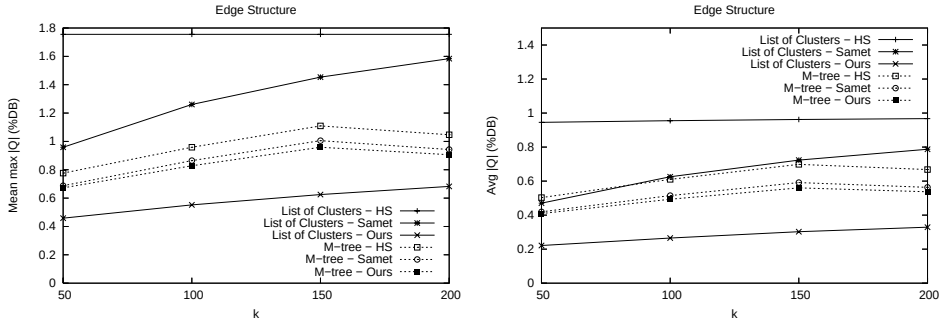

Figure 9: Edge structure: Mean of the maximum queue length (left) and average queue length (right).

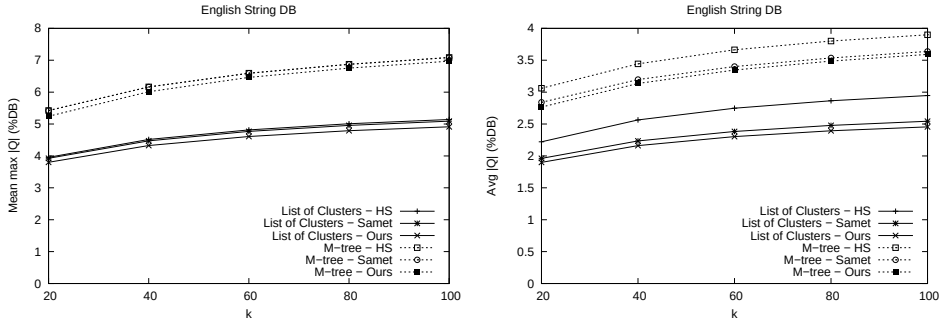

Figure 10: English String DB: Mean of the maximum queue length (left) and average queue length (right).

Table 2: Storage requirements (maximum and average queue length) of our algorithm (standard algorithm: 100%, $k = 50$)

Database	LC-max	LC-avg	MT-max	MT-avg
Gaussian 8-D	49.4%	48.3%	93.7%	86.5%
Gaussian 16-D	19.5%	19.2%	96.2%	88.3%
Gaussian 32-D	18.5%	18.2%	96.3%	89.1%
Color Histogram	34.6%	32.3%	93.4%	89.1%
Layout Histogram	30.8%	28.4%	93.7%	88.9%
Edge Structure	26.1%	23.4%	86.8%	81.3%
English String DB ($k = 100$)	95.6%	83.4%	98.4%	92.1%

bubbles, which are "abstract" index nodes with no elements inside. Bubbles can be used to prune index nodes from the active page list using the distance estimators, even if the algorithm has not yet visited the index nodes which actually contain the objects inside the bubble. We tested our algorithm with several synthetic and real-world datasets, using two state-of-art index structures for metric spaces. Our experimental results confirm that the storage requirements of our proposed algorithm are considerably smaller compared with the standard k-NN algorithm (up to 5 times smaller).

We observed that the best results (i.e., the larger savings in space) were obtained with the List of Clusters. This result can be explained as follows: the List of Clusters produces more compact balls (regions) than the M-tree, due to the dynamic nature of the latter. For example, for the color histogram database the average covering radius using the M-tree was 0.32, while the average covering radius using List of Clusters was only 0.18. Also, the M-tree creates more balls than the List of Clusters, and not all them are necessarilly full (as in the case of the List Clusters). This means that the "density" of each ball in the M-tree is smaller than in the balls of the List of Clusters. Thus, the search algorithm is able to find better distance estimations to the balls with the List of Clusters, which allows this index to discard balls from Q sooner than the M-tree.

We plan to continue exploring the trade-off between the index size and the storage requirements for the active page list. Further improvements on the average length of Q may be obtained if one has more structural information about the balls, at the cost of storing more information on each index node.

Although we focused in this paper on indexes for metric spaces, our technique is general and can be adapted with minimal effort to be used with spatial access methods. In that case, each subtree is usually bounded by a hyperrectangle. Our heuristic translates into the following rule: *Assume a tree node contains bsize elements within a hyperrectangle. Then there are bsize elements at distance* $\max \delta(q, c)$*, where c ranges among all the corners of the hyperrectangle.* This heuristic is different from the usual MinMaxDist [18], which gives a better distance estimator but holds only for one object per tree node. If we use the simpler rule by Samet [19] our work builds on, and translate it to a spatial data structure, the result is always inferior to the MinMaxDist heuristic.

Acknowledgments

We thank Christian Rohrdantz for implementing the algorithms and the index structures, and for running the experimental evaluation.

References

[1] M. Baroni, G. Cruciani, S. Sciabola, F. Perruccio, and J. Mason. A common reference framework for analyzing/comparing proteins and ligands. Fingerprints for ligands and proteins (FLAP): Theory and applications. *Journal of Chemical Information Modeling*, 47:279–294, 2007.

[2] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[3] B. Bustos, D. Keim, D. Saupe, T. Schreck, and D. Vranić. An experimental effectiveness comparison of methods for 3D similarity search. *International Journal on Digital Libraries, Special issue on Multimedia Contents and Management in Digital Libraries*, 6(1):39–54, 2006.

[4] E. Chávez and G. Navarro. A compact space decomposition for effective metric indexing. *Pattern Recognition Letters*, 26(9):1363–1376, 2005.

[5] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.

[6] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. 23rd Intl. Conf. on Very Large Databases (VLDB'97)*, pages 426–435. Morgan Kaufmann, 1997.

[7] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. D-index: Distance searching index for metric data sets. *Multimedia Tools and Applications*, 21(1):9–33, 2003.

[8] T. Funkhouser, M. Kazhdan, P. Shilane, P. Min, W. Kiefer, A. Tal, S. Rusinkiewicz, and D. Dobkin. Modeling by example. *ACM Transactions on Graphics*, 23(3):652–663, 2004.

[9] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[10] S. Hettich and S. Bay. The UCI KDD archive [http://kdd.ics.uci.edu], 1999.

[11] A. Hinneburg, C. Aggarwal, and D. Keim. What is the nearest neighbor in high dimensional spaces? In *Proc. 26th International Conference on Very Large Databases (VLDB'00)*, pages 506–515. Morgan Kaufmann, 2000.

[12] G. Hjaltason and H. Samet. Ranking in spatial databases. In *Proc. 4th Intl. Symp. on Advances in Spatial Databases*, LNCS 951, pages 83–95. Springer-Verlag, 1995.

[13] G. Hjaltason and H. Samet. Incremental similarity search in multimedia databases. Technical Report CS-TR-4199, University of Maryland, Computer Science Department, 2000.

[14] G. Hjaltason and H. Samet. Index-driven similarity search in metric spaces. *ACM Trans. on Database Systems*, 28(4):517–580, 2003.

[15] Daniel A. Keim. Efficient geometry-based similarity search of 3D spatial databases. In *Proc. ACM International Conference on Management of Data (SIGMOD'99)*, pages 419–430. ACM Press, 1999.

[16] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[17] G. Navarro. Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46, 2002.

[18] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. ACM International Conference on Management of Data (SIGMOD'95)*, pages 71–79. ACM Press, 1995.

[19] H. Samet. Depth-first k-nearest neighbor finding using the MaxNearestDist estimator. In *Proc. 12th Intl. Conf. on Image Analysis and Processing (ICIAP'03)*, pages 486–491. IEEE Computer Society, 2003.

[20] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[21] R. Santos-Filho, A. Traina, C. Traina Jr., and C. Faloutsos. Similarity search without tears: The OMNI family of all-purpose access methods. In *Proc. 17th Intl. Conf. on Data Engineering (ICDE'01)*, pages 623–630. IEEE Computer Society, 2001.

[22] J. Uhlmann. Implementing metric trees to satisfy general proximity/similarity queries. Manuscript, 1991.

[23] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.