An Effective Clustering Algorithm to Index High Dimensional Metric Spaces*

Edgar Chávez
Escuela de Ciencias Físico-Matemáticas
Universidad Michoacana
Edificio "B", Ciudad Universitaria
Morelia, Mich. México 58000
elchavez@zeus.ccu.umich.mx

Gonzalo Navarro
Department of Computer Science
University of Chile
Blanco Encalada 2120, Santiago, Chile
gnavarro@dcc.uchile.cl

Abstract

A metric space *consists of a collection of objects and a* distance function *defined among them, which satisfies the triangular inequality. The goal is to preprocess the set so that, given a set of objects and a query, retrieve those objects close enough to the query. The number of distances computed to achieve this goal is the complexity measure. The problem is very difficult in the so-called* high-dimensional *metric spaces, where the histogram of distances has a large mean and a small variance. A recent survey on methods to index metric spaces has shown that the so-called* clustering *algorithms are better suited than their competitors,* pivot-based *algorithms, to cope with high-dimensional metric spaces. In this paper we present a new clustering method that achieves much better performance than all the existing data structures. We present analytical and experimental results that support our claims and that give the users the tuning parameters to make optimal use of this data structure.*

1. Introduction

The concept of "proximity" searching has applications in a vast number of fields. Some examples are non-traditional databases (where the concept of exact search is of no use and we search for similar objects, e.g. databases storing images, fingerprints or audio clips); 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); computational biology (where we want to find a DNA or protein sequence in a database allowing some errors due to typical variations); function prediction (where we want to search the most similar behavior of a function in the past so as to predict its probable future behavior); etc.

All those applications have some common characteristics. There is a universe $\mathbb{X}$ of *objects*, and a non-negative *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*

$$\begin{aligned} d(x,y) &= 0 \quad \Leftrightarrow \quad x = y \\ d(x,y) &= d(y,x) \\ d(x,z) &\leq d(x,y) + d(y,z) \end{aligned}$$

where the last one is called the "triangular inequality" and is valid for many reasonable similarity functions. The smaller the distance between two objects, the more "similar" they are. This distance is considered expensive to compute (think, for instance, in comparing two fingerprints). We have a finite *database* $\mathbb{U} \subseteq \mathbb{X}$, 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 (a *query* q), we must retrieve all similar elements found in the database. There are three typical queries of this kind:

Range queries: retrieve all elements which are within distance r to q.
That is, $(q,r) = \{u \in \mathbb{U} \ / \ d(u,q) \leq r\}$.

*This project has been partially supported by CYTED VII.13 AMYRI Project (both authors), CONACyT grant R-28923A (first author) and FONDECYT Project 1-000929 (second author).

Nearest neighbor (NN) queries: retrieve the closest elements to q in $\mathbb{U}$.
That is, $nn(q) = \{u \in \mathbb{U} \ / \ \forall v \in \mathbb{U},\ d(u,q) \leq d(v,q)\}$.

k-NN queries: retrieve the k closest elements to q in $\mathbb{U}$.
That is, retrieve a set $nn_k(q) \subseteq \mathbb{U}$ such that $|nn_k(q)| = k$ and $\forall u \in nn_k(q), v \in \mathbb{U} - nn_k(q), d(u,q) \leq d(v,q)$.

Given a database of $|\mathbb{U}| = n$ objects, all those queries can be trivially answered by performing n distance evaluations. Since the distance function is assumed to be expensive to compute, the goal is to structure the database so that we perform few distance evaluations. All the existing techniques work by discarding elements using the triangular inequality.

We concentrate in range queries in this paper, as the others can be systematically built over these [10]. The set of points of $\mathbb{X}$ that are at distance at most r to q is called the "query ball", so (q,r) is the intersection of the query ball and $\mathbb{U}$.

A particular case of this problem arises when the space is $\mathbb{R}^k$. There are effective methods for this case, such as kd-trees [3] or R-trees [13]. However, for more than roughly 20 dimensions those structures cease to work well. We focus in this paper in general metric spaces, although the solutions are well suited also for k-dimensional spaces.

It is interesting to notice that the concept of "dimensionality" can be translated to metric spaces as well: the typical feature in high dimensional spaces is that the probability distribution of distances among elements has a very concentrated histogram (with larger mean as the dimension grows), hampering the work of any similarity search algorithm [5, 7]. In the extreme case we have a space where $d(x,x) = 0$ and $\forall y \neq x,\ d(x,y) = 1$, where it is impossible to avoid a single distance evaluation at search time. We say that a general metric space is high dimensional when its histogram of distances is concentrated. We use in this paper a quantitative definition of the intrinsic dimensionality proposed in [10]:

Definition: The *intrinsic dimensionality* of a metric space is defined as $\rho = \frac{\mu^2}{2\sigma^2}$, where μ and σ^2 are the mean and variance of its histogram of distances.

Under this definition, a random vector space with k coordinates has intrinsic dimension $\Theta(k)$, so the definition extends naturally that of vector spaces.

In the same survey [10] a number of approaches to solve the problem of proximity searching in general metric spaces are considered, which are divided in two classes:

- *Pivot-based algorithms:* which select a number of "pivots" from the database and classify all the other elements according to their distances to the pivots. The distances between elements and pivots and between the query q and the pivots are used together with the triangular inequality to filter out elements of the database without actually measuring their distance to q. These algorithms generally improve as more pivots are added, although the space requirements of the indexes increase as well.

- *Clustering algorithms:* which divide the set into spatial zones which are as compact as possible, and are able to discard complete zones by performing few distance evaluations (e.g. between the query q and a centroid of the zone). The partition into zones can be hierarchical, but the indexes use a fixed amount of memory and cannot be improved by giving them more space.

As shown in [10], clustering algorithms deal better with high dimensional metric spaces. Despite that pivot-based algorithms can improve by using more memory, they need more and more memory to beat clustering algorithms as the dimension grows. For intrinsic dimension around 20 they already need impractical amounts of extra space. Therefore, clustering algorithms seem a promising alternative to index high dimensional metric spaces.

In this paper we present a new clustering algorithm based on an asymmetrical querying process. We present analytical results based on the intrinsic dimension to analyze its different alternatives and tuning options, and later we present experimental results showing that it outperforms by far all the existing schemes.

2. Related Work

Different data structures have been proposed to filter out elements based on the triangular inequality (see [10] for a complete survey). We divide the exposition according to the two classes of techniques.

2.1. Pivot-based Algorithms

Burkhard-Keller Trees (bk-trees) [6] are designed for discrete distance functions: they select a pivot element p as the root of the tree, and put at child i the elements which are at distance i to the pivot. Each subtree is recursively built with the same technique until there

are b elements or less, in which case the elements are simply stored in a "bucket" at the tree leaf. A range query q with tolerance radius r is searched by measuring $d(p,q)$, reporting p if appropriate, and entering only into subtrees numbered $d(p,q)-r$ to $d(p,q)+r$. The rest need not be considered because of the the triangle inequality. The buckets reached are exhaustively compared against q.

Fixed Queries Trees (fq-trees) [2] are an evolution where the same pivot is used for all the nodes of the same level of the tree. In this case the pivot does not need to belong to the subtree. Many comparisons are saved in the backtracking process because only one different pivot per level exists. However, the tree is taller. A variant called Fixed Height fq-tree (fhq-tree) is also proposed where all the leaves are at the same depth h, regardless of the bucket size.

Vantage Point Trees (vp-trees) [20, 22] are designed for continuous distance functions. The root has two equal-size subtrees that divide the elements in closer to and farther from the root. This can be extended to m-ary trees (mvp-trees) [5, 4].

Finally, algorithms like AESA [21], LAESA [16, 15] and its variants [18, 8] and Fixed Queries Arrays (fq-arrays [9]) are based in a common idea: k pivots are selected and each object is mapped to k coordinates which are its distances to the pivots. Later, the query q is also mapped and if it differs from an object in more than r along some coordinate then the element is filtered out by the triangle inequality. That is, if for some pivot p_i and some element v of the set it holds $|d(q,p_i)-d(v,p_i)|>r$, then we know that $d(q,v)>r$ without need to evaluate $d(v,q)$. The elements that cannot be filtered out using this rule are directly compared.

An interesting feature of most of these algorithms is that they can reduce the number of distance evaluations by increasing the number of pivots. Define $D_k(x,y) = \max_{1\le j\le k}|d(x,p_j)-d(y,p_j)|$. Using the pivots $p_1,...,p_k$ is equivalent to discarding elements u such that $D_k(q,u)>r$. As more pivots are added we need to perform more distance evaluations (exactly k) to compute $D_k(q,*)$, but on the other hand $D_k(q,*)$ increases its value and hence it has a higher chance of filtering out more elements. It follows that there exists an optimum k, This optimum, however, cannot be normally reached because it is too high in terms of space requirements: kn distances have to be precomputed and stored in order to use k pivots. Hence, in general these methods use as many pivots as they can, and they are normally well below their optimum.

2.2. Clustering Algorithms

Generalized Hyperplane Trees (gh-trees) [20] use two "centers" for each tree node and divide the space according to which of the two centers is closer to each object. At search time the query enters into the subtrees whose zone of influence has a nonempty intersection with the query ball.

Bisector Trees [14, 19] are similar but the zones are not defined according to which is the closest center but using the concept of "covering radius". The *covering radius* of a zone is the minimum radius of a sphere that is necessary to contain all the points in the zone, and the elements are inserted in the subtrees trying to minimize covering radii. This is generalized to Voronoi Trees (v-trees) in [12] to reduce more the covering radii.

Gh-trees are generalized to an m-ary partition in the Geometric Near-neighbor Access Tree (gna-tree) [5], which makes a Voronoi-like partition of the space [1] among the m pivots at each node of the tree. However, the gna-tree uses also the covering radius criterion to prune the search even more.

The M-tree [11] also takes m elements and divides the space among its zones of influence, but it uses only the covering radius information to classify and search the elements. The M-tree is able of dynamic insertion and deletion of points and is optimized for secondary memory.

Spatial Approximation Trees (sa-trees) [17] are based on approaching the query spatially: the search starts at the root of the tree and moves to neighbors that are closer to the query. The ideal data structure to obtain this is a Voronoi graph, which in the paper is proven impossible to build on a general metric space. Therefore the sa-tree is a simplification which forces some backtracking in the tree.

3. A New Clustering Technique

We propose now a novel technique to index a metric space. We start by taking a "center" $c\in\mathbb{U}$ and a radius r (whose value is discussed later). We define the *center ball* of (c,r) (or just c if no ambiguity is possible) as the subset of elements of $\mathbb{X}$ which are at distance at most r from c. Now we define

$$I_{\mathbb{U},c,r} \;=\; \{u\in\mathbb{U}-\{c\},\ d(c,u)\le r\}$$

as the bucket of "internal" elements, which lie inside center ball of c, and

$$E_{\mathbb{U},c,r} \;=\; \{u\in\mathbb{U},\ d(c,u)>r\}$$

as the rest of the elements (the "external" ones). Now the process is repeated recursively inside E. The construction procedure returns a list of triples (c_i, r_i, I_i) (center,radius,bucket) and it is shown in Figure 1.

```
Build (𝕌)
    if 𝕌 = ∅ then return an empty list
    Select c ∈ 𝕌
    Select a radius r
    I  ⟵  {u ∈ 𝕌 − {c},  d(c,u) ≤ r}
    E  ⟵  𝕌 − I
    return (c,r,I):Build(E)
```

Figure 1. The construction algorithm. The operator ":" is the list constructor. It is not hard to remove the tail recursion to make it iterative.

The data structure that is built looks rather symmetric, but it is not. The first center chosen has preference over the subsequent centers in case of overlapping balls. Figure 2 illustrates. All the elements that are inside the ball of the first center (c_1 in the figure) are stored in its I bucket, despite that they may also lie inside the I buckets of subsequent centers (c_2 and c_3 in the figure). The figure also shows how the data structure can be seen as a list.

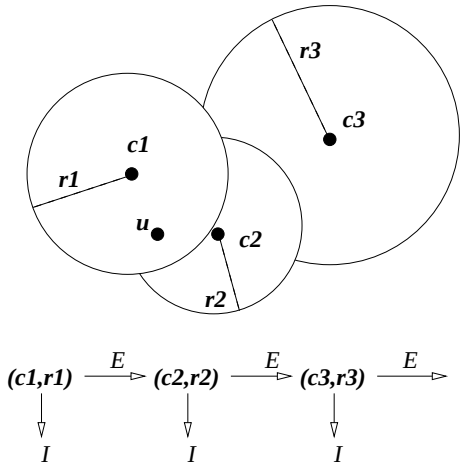

Figure 2. The influence zones of three centers taken in this order: c_1, c_2, c_3. We also show a list arrangement for the data structure.

The search algorithm is depicted in Figure 4. The idea is that if the first center chosen is c and its radius is r_c, then the search for a query (q, r) starts by measuring $d(c, q)$ and adding c to the set of results if appropriate. Then, we search exhaustively inside the bucket I only if the query ball has some intersection with the center ball of c. Now, given the asymmetry of the data structure, we can also prune the search in the other direction: if the query ball is totally contained in the center ball of c, then we do not need to consider E because by construction we know that all the elements that are inside the query ball have been inserted in I.

```
Search (L,q,r)
    if L is empty then return
    Let L = (c, r_c, I) : E
    Compute d(c,q)
    if d(c,q) ≤ r then add c to the results
    if d(c,q) ≤ r_c + r then search I exhaustively
    if d(c,q) > r_c − r then Search (E,q,r)
```

Figure 3. The search algorithm. It is not hard to remove the tail recursion to make it iterative.

This is an essential feature absent in other clustering algorithms, where the search needs to enter into all the clusters which are intersected by the query ball. With our data structure the consideration of the relevant clusters can be preempted as soon as the query ball is totally contained in a cluster. Figure 4 illustrates.

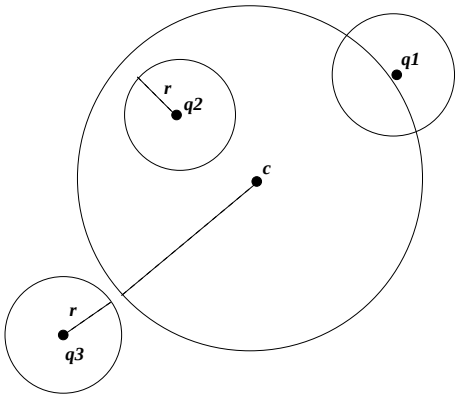

Figure 4. Three cases of query ball versus center ball. For q_1 we need to consider the current bucket and the rest of centers. For q_2 we can prune the search inside the rest of the clusters. For q_3 we can avoid considering the current bucket.

Compared to other clustering algorithms, ours uses only the covering radius criterion (and not Voronoi-like areas), but it is able to prune more by using the order of the centers, as explained. It is also possible to see our list of clusters as a particular case of a vp-tree or an M-tree by considering I and E as the left and right subtrees of the root c, but there is a fundamental difference that is made clear in the next section: while those data structures try to build balanced trees, ours is extremely unbalanced, as I is much smaller than E. Moreover, our I bucket does not have any internal structure.

4. Analysis

The description of the data structure does not specify how the center and the radius are selected at each point of the construction algorithm. As this is related to the efficiency and not the correctness of the data structure, we have left it unspecified until now, when the efficiency is analyzed.

Let us consider the histogram of distances between arbitrary elements of $\mathbb{X}$. As implied by the definition of the intrinsic dimensionality, this histogram is compressed and moved to the right as the dimension of the metric space grows.

When we choose a random center c, the histogram of distances to c is similar to the global histogram. Therefore, we can use the global histogram to consider the effect of radius selection. Moreover, we assume that the histogram remains unchanged after removing the elements corresponding to each cluster. All these are reasonable simplifications.

4.1. Clusters of Fixed Radius

The simplest alternative seems to be selecting a fixed radius $r_i = r^*$ for all the clusters in the list. This implies that a fixed proportion p^* of the remaining elements lie inside the center ball, which corresponds to the mass of the histogram in the interval $[0, r^*]$. Figure 5 (left) illustrates.

Let us call p_+ the mass of the histogram in the interval $(r^*, r^* + r]$ and p_- that in the interval $[r^* - r, r^*)$. Then $p_I = p^* + p_+$ is the probability that a given I bucket has to be examined, while $p_E = 1 - (p^* - p_2)$ is the probability of having to continue considering the other buckets (see the right plot of Figure 5). In a real application these probabilities can be estimated with a Monte Carlo method. Furthermore, the average number of elements in the i-th cluster is $m_i = np^*(1-p^*)^{i-1}$, which decreases as we advance in the list.

The average search cost can be computed as follows. We pay one comparison against the first center and with probability p_I we have to consider the first bucket, which has on average $m_1 = p^* n$ elements. With probability p_E we continue considering the rest of the buckets, which arrange $(1-p^*)n$ elements on average. This yields an average search cost of $C(n)$, where

$$C(n) \;=\; 1 \;+\; np^* p_I \;+\; p_E C((1-p^*)n)$$

$$= \;\frac{np^*(p^* + p_+)}{1-(1-p^*)(1-p^*+p_-)} \;+\; \frac{1}{p^* - p_-}$$

We would like to find the optimum p^*. Under the simplifying assumption that the search radius is zero (which implies $p_+ = p_- = 0$), the above cost is minimized for

$$p^* \;=\; \frac{2}{1+\sqrt{2n}} \;\approx\; \sqrt{\frac{2}{n}}$$

and the corresponding search cost is $1/2 + \sqrt{2n} \approx \sqrt{2n}$. The expected length of the list (solving $(1-p^*)^h n = 1$) is $\log_{1/(1-p^*)} n \approx \sqrt{n/2} \ln n$ buckets of size $\sqrt{2n}$ in the first buckets and decreasing as we advance in the list.

The solution for the general case depends, unfortunately, on p_+ and p_-, which in turn depend on the query radius r. As this cannot be determined beforehand, one has to optimize the structure for a given r or to use a simplification as the one we have done.

4.2. Clusters of Fixed Size

Another choice is to try to have a fixed number m^* of elements inside each center ball, and to define r_i accordingly. This also fixes the length of the list to $\lceil n/(m^*+1) \rceil$.

When we are building the i-th cluster the number of remaining elements is $n - m(i-1)$. This means that we have to select the radius r_i so that $p_i = m^*/(n - m^*(i-1))$ of the mass of the histogram lies in the interval $[0, r_i]$. Compared to the previous approach, we now increase the radius instead of letting the number of elements of the clusters reduce as we advance in the list.

The average search cost can be computed as before. We pay one comparison against the first center and with probability $p_{I,1} = p_1 + p_+$ we have to consider the first bucket, which has m elements. With probability $p_{E,1} = 1 - (p_1 - p_-)$ we continue considering the rest of the buckets, which arrange $n - m^*$ elements. This yields an average search cost of $C(1)$, where

$$C(i) \;=\; 1 \;+\; m^* p_{I,i} \;+\; p_{E,i} C(i+1)$$

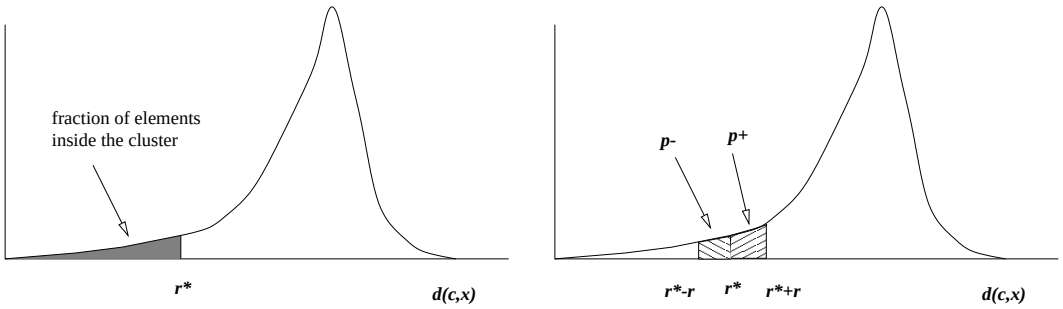

Figure 5. The histogram of distances between c and a random element x. On the left, the grayed part is the fraction of the set captured by a ball of radius r centered at c. On the right we have plotted the areas corresponding to p_+ and p_-.

$$= \sum_{i \geq 0} (1 + m^* p_{I,i+1}) \prod_{1 \leq j \leq i} p_{E,j}$$

Again we make the simplification of assuming $r = 0$ and therefore $p_+ = p_- = 0$. This makes

$$\prod_{1 \leq j \leq i} p_{E,j} = \prod_{1 \leq j \leq i} 1 - p_j = \frac{n - im^*}{n}$$

and the whole cost formula becomes

$$\frac{n(m^*+2)}{2(m^*+1)^2} + \frac{2m^{*2} + m^* + 2}{2(m^*+1)} + \frac{m^{*2}}{n} \approx \frac{n}{2m^*} + m^*$$

where the first term corresponds to the expected comparisons against the centers and the second to those inside the buckets. This cost is optimized for $m^* = \sqrt{n/2}$, where the expected search cost is about $\sqrt{2n}$, independent on the intrinsic dimension of the space.

In both cases we have obtained a list of length about $\sqrt{n}$ with about $\sqrt{n}$ elements in each bucket and about $\sqrt{n}$ search cost. The main difference is in the exact form of the list. As the analysis does not tell which is better, we have to decide that experimentally.

Another thing that the analysis does not tell is how the dimension affects the search times. As the dimension grows, p_+ and p_- grow for a given r, and therefore the probability of entering into more clusters increases. This will be measured experimentally.

4.3. Center Selection

Independently on how we select the radius of each cluster, we can apply different heuristics to select the i-th center. Some choices are:

($p1$) At random.

($p2$) The element closest to c_{i-1} in the remaining set.

($p3$) The element farthest from c_{i-1} in the remaining set.

($p4$) The element minimizing the sum of distances to previous centers.

($p5$) The element maximizing the sum of distances to previous centers.

The first alternative is the simplest but not necessarily the best one. The second one aims at building a bucket ordering that moves slowly across the metric space. The third one aims at minimizing the overlap between clusters. ($p4$) and ($p5$) are more global versions of ($p2$) and ($p3$), respectively. Moreover, ($p2$) and ($p4$) aim at finding a next pivot close to the current one, as in sa-trees, while ($p3$) and ($p5$) try that the volumes of different clusters do not overlap, as gna-trees.

5. Building and Updating the Data Structure

5.1. Construction

Our data structure can be built by brute force in $O(n^2/p^*)$ time for fixed radius clusters and $O(n^2/m^*)$ time for fixed size clusters. Using the optimal settings this is $O(n^{3/2})$ in both cases.

This cost is independent on the dimension, and can be reduced by noting that I is defined as the result of a range query (c_i, r^*) for fixed radius clusters and of a nearest neighbor $nn_{m^*}(c_i)$ query for fixed size clusters. Therefore, another (cheaper to build) data structure built on the metric space could be used as an auxiliary data structure to build ours. This matches especially well with the center selection heuristics ($p1$) and ($p2$),

while the others may need extra work. It is also worthwhile to note that this auxiliary data structure should be able of efficient deletion of the elements that are inserted into each new cluster, in order to answer queries on the remaining set.

The fixed radius data structure has the disadvantage that the bucket sizes cannot be predicted in advance, which complicates a bit secondary memory arrangements. On the other hand, updating the structure is simpler than with fixed size buckets.

5.2. Updating

Let us consider the process of inserting a new element in the fixed radius data structure. If p^* (and r^*) have been correctly computed in the beginning, they should not change as we insert more elements, and the insertion should be done by traversing the list of clusters until the element falls inside some center ball, or otherwise creating a new cluster for it at the end of the list.

Deletion can be trivially done except if a center is deleted, in which case a first choice is to keep it anyway as a fake element. A safer choice is to remove the whole bucket from the list and reinsert all the elements (note that the insertion of those elements can be done just in the tail of the list, as we already know that they do not lie inside any previous center ball).

However, a massive insertion of elements may affect the optimality of the r^* value chosen (e.g. $\sqrt{n}$). In those cases a periodic rebuild of the whole data structure may be benefical for the performance.

These update operations are a bit more complex if we have a fixed bucket size. When inserting an element, as soon as we find its appropriate ball i, the bucket will overflow. Hence we take the element of the bucket which is farthest from the center c_i, remove it from the bucket (modifying r_i accordingly), and continue the insertion process in the tail of the list with the new element. Hence we are guaranteed to traverse the whole list of centers for every insertion. Deletion presents a more difficult problem, since the bucket underflows and we have to find the next nearest neighbor of c_i in the rest of the elements. This can be done using the same data structure, but it is costly anyway. Two choices are lazy deletion (i.e. leave the whole hoping that a new insertion will fit the place) and setting a range of values for m^* instead of a fixed value. Deletion of a center can be handled as for the fixed radius data structure.

5.3. Secondary Memory

Our data structure has the advantage of a rather predictable access pattern. The cluster centers are compared always in the same order. Sometimes we need to retrieve a whole bucket, sometimes not. Finally, we can stop the search at any moment in the list of centers.

A simple linear arrangement of the centers yields an efficient disk layout for this search algorithm, with minimal seek time. The buckets should be similarly arranged in a separate list. Fixed size buckets make this extremely simple, while fixed radius clusters need an expansion mechanism to accommodate their varying size. There are well known mechanisms of that type, and the histogram can be used as a tool to upper bound overflow probabilities.

6. Experimental Results

We present now experiments that help determining the best choice for our data structure and comparing it against previous work.

Our metric space is the unitary real cube in k dimensions ($[0,1)^k$) under the Euclidean distance. We generate a fixed number n of random points and search random queries q with a radius r such that 0.01% to 0.1% of the set of points is retrieved. We show the results as a function of the dimension k of the space. Despite that this is a restricted case of vector space, we can in this case effectively control the dimension, which is difficult to do in real-world examples. We make the experiments with $n = 100,000$ elements.

We have made all the experiments on both versions of the algorithms: fixed bucket size and fixed radius. As we show in the last experiments, the former turned out to be superior, so for the rest of the experiments we show only the results on fixed bucket size.

6.1 Tuning Our Data Structure

Our first experiment tries to determine the best choice among $(p1)-(p5)$. Figure 6 shows the results using two different choices for m^* (12 and 100). It can be seen that $(p3)$ and $(p5)$ are better choices, which favors heuristics that try to minimize the intersection among clusters [22, 5]. The difference among $(p3)$ and $(p5)$ is not statistically significant when using a large bucket size. With a smaller bucket size (12) the $(p5)$ heuristics is clearly better and therefore we use $(p5)$ from now on, as it is a more elaborated version of $(p3)$ that should work in more complex scenarios (e.g. clustered data).

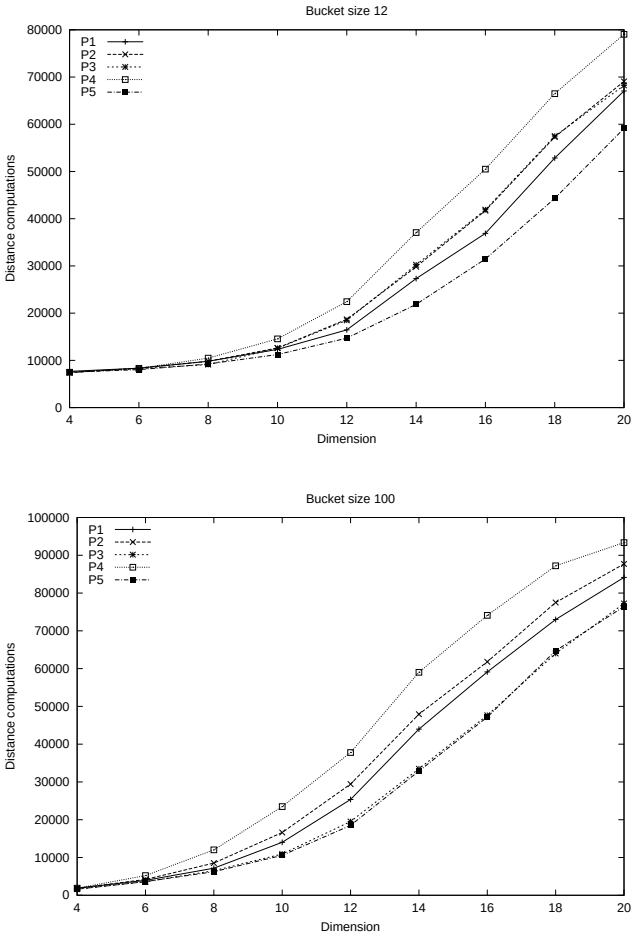

Figure 6. Number of distance evaluations for pivot selection techniques ($p1$) to ($p5$), as the dimension grows. We show fixed bucket sizes $m^* = 12$ (top) and $m^* = 100$ (bottom), capturing 0.01% of a database of size $n = 100,000$.

We now focus on the optimal m^*. Figure 7 shows that the optimal optimal value depends on the dimension, starting at $m^* = 200$ for low dimension and ending at $m^* = 6$ for high dimensions. The growth of the optimal search cost as the dimension increases is not so sharp as in most of the previous work (we compare later the different algorithms).

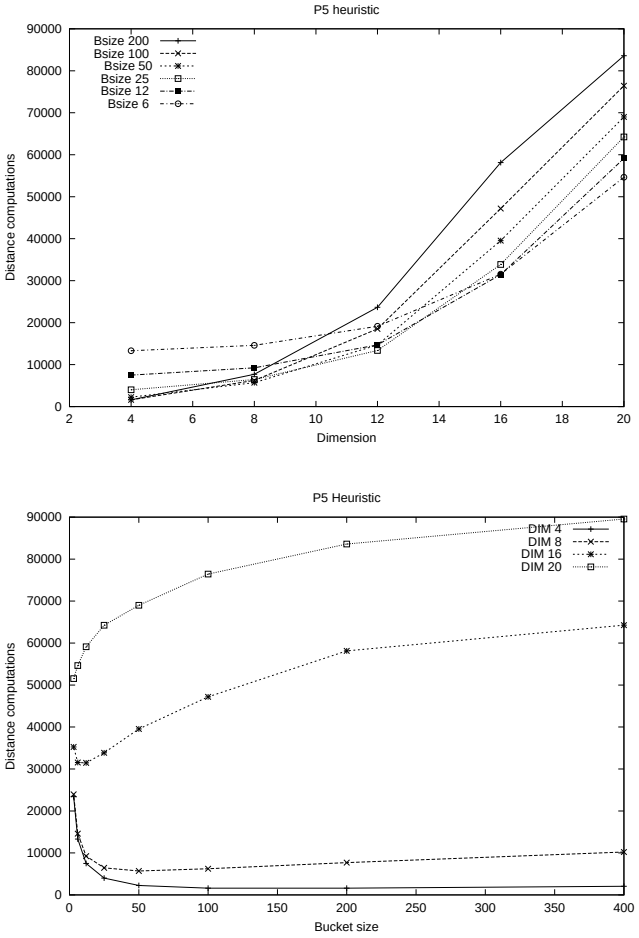

Figure 7. On the top, number of distance evaluations for different bucket sizes as a function of the dimension. On the bottom, search cost as a function of the bucket size for different dimensions. The query captures 0.01% of a database of $n = 100,000$ elements.

The analysis predicts that the optimal bucket size also depends on the database size, giving $\sqrt{n}$ as a lower bound for the search cost. Figure 8 shows this fact. The first plot shows that there is a dependence (on fixed dimension 8), while the second plot shows the

cost when the optimum bucket size is used for each dimension and each database size.

Using least squares we find that $\sqrt{n}$ is a reasonable lower bound (recall that it was obtained by assuming a search with radius zero). Retrieving 0.01% of the database, we get search complexities $O(n^{0.65})$ for dimension 4, $O(n^{0.69})$ for dimension 8, $O(n^{0.72})$ for dimension 12, $O(n^{0.81})$ for dimension 16 and $O(n^{0.88})$ for dimension 20. The relative error of these approximations is around 5%.

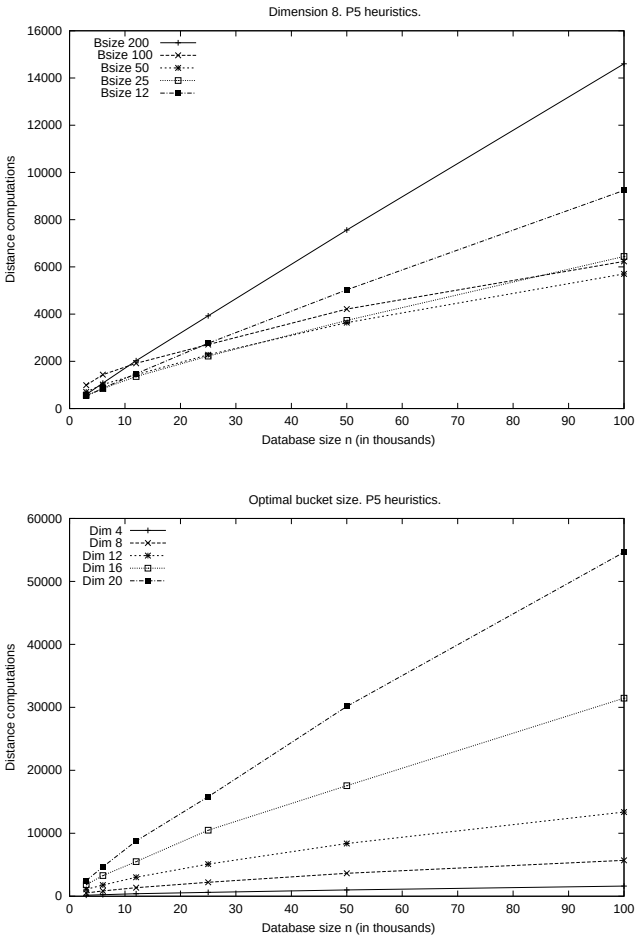

Figure 8. On the top, number of distance evaluations for different bucket sizes as a function of the database size, in 8 dimensions. On the bottom, search cost when the optimal bucket size is used for each dimension and database size. The query captures 0.01% of n.

Just to check the accurateness of the analysis, we have run experiments with search radius zero. According to the analysis, the search cost with the optimal bucket size is $O(\sqrt{n})$ independently of the dimension. Figure 9 shows the search time in 8 and 20 dimensions using the optimal bucket size. The figure makes it clear that there is no dependence on the dimension in this case. Least squares yields $O(n^{0.49})$ and $O(n^{0.51})$, exactly as predicted.

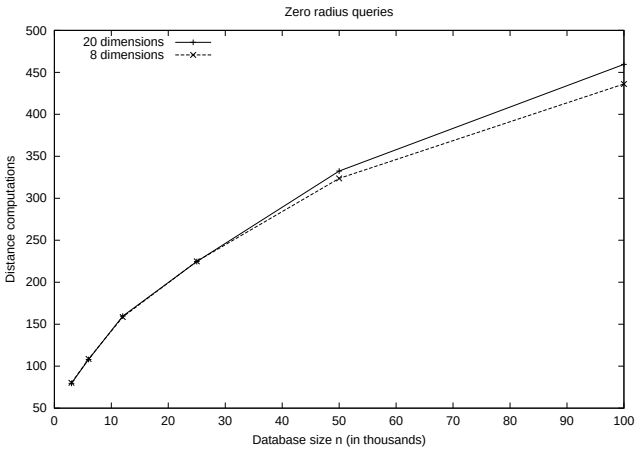

Figure 9. Number of distance evaluations as n grows for different dimensions using optimal bucket size and search radius zero.

6.2 Comparing Against the Rest

We compare now our data structure against some existing techniques. We have included our fixed bucket size and fixed radius alternatives, both using their optimal setups, to show that fixed bucket size outperforms the other. In particular, the P5 heuristics turned out to be the best one for fixed radius as well, the optimal radius size moving from 2.5 to 1.0 times the query radius as the dimension moves from 4 to 20.

Observe in Figure 10 that two pivot-based algorithms (fq-arrays and LAESA) have needed at least 32 and 64 times more memory, respectively, than the other clustering algorithms (gna-trees and sa-trees) in order to beat them in high dimension. This evidence favors the use of clustering algorithms instead of pivot-based ones in high dimensions. Beating clustering algorithms with pivot based ones becomes be even more difficult for higher dimensions and bigger search radius.

As the other clustering algorithms, ours does not need more memory to cope with higher dimensions. Moreover, the search complexity of our new schemes grows much slower as the dimension grows. In particular, the combination we have chosen is by far the best

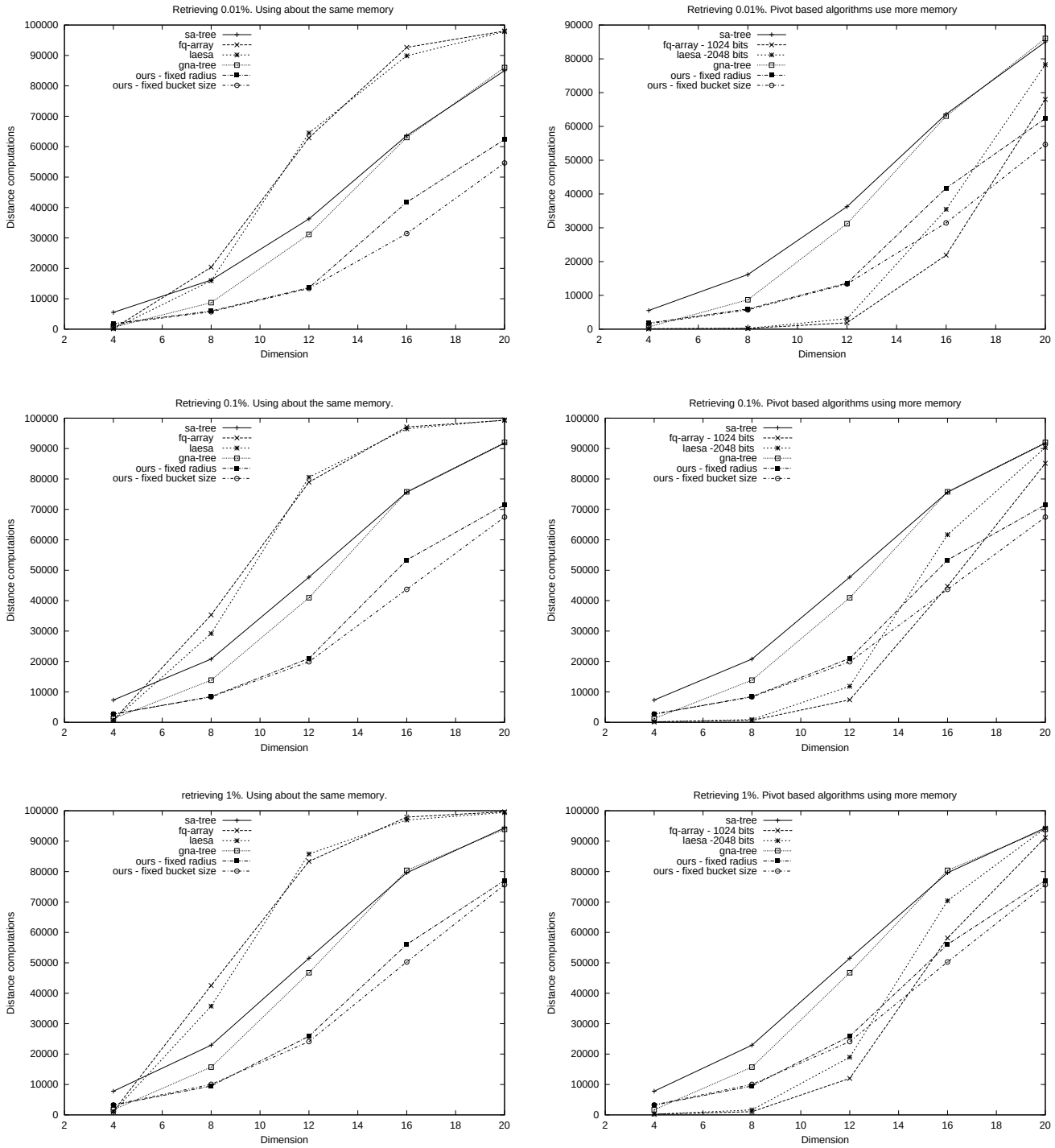

Figure 10. Comparison with existing approaches for varying dimension. On the left all the algorithms use (about) the same memory, on the right the pivot-based algorithms are allowed to use more memory (indicated in bits per database element). The query captures 0.01% (first row), 0.1% (second row) and 1% (third row) of a database of $n = 100,000$ elements.

in 20 dimensions, even if we allow using 32 times more memory to competing pivot based algorithms. For example, with $n = 100,000$ and 20 dimensions we inspect only 55% of the set to retrieve 0.01% of the set, against almost 70% for the best competitor.

The situation improves for us even more as n grows. Figure 11 shows the behavior of all the algorithms for increasing n. As can be seen, our algorithms present a much slower growing search time than all previous work, especially on high dimensions. Still the fixed bucket variant ourperforms the fixed radius one in high dimension.

7. Conclusions

We have presented a new clustering algorithm which is experimentally shown to be much more efficient than the others, especially in high dimensions. We have found analytical recommendations to tune the data structure, despite that the fine tuning still has to be empirical. With respect to the database growth, our analysis predicts a sublinear behavior, which we experimentally verified to have a form close to $O(n^\alpha)$, for some $0.5 < \alpha < 1$. We have also shown experimentally that the search cost of our algorithms grows much slower than the others with respect to the database size.

Among all the strategies we have tested, the fixed bucket size which selects the next pivot far away from previous ones is simple to tune, behaves much better than its competitors, and is well suited for secondary memory implementations.

An interesting alternative view of our data structure is that each cluster representative is in fact a pivot, and the only information we store for each pivot and each database element (in fact not for all pairs) is whether the distance among them is smaller or larger than a given threshold. Moreover, the list of clusters permits storing this information in a compact form. The key of its success is that this compact representation permits having a huge number of pivots with constant space per element. This is impossible with traditional pivot based schemes.

Future work involves improving the construction procedure, possibly by using auxiliary data structures to build the I buckets. We also plan to pursue in the problem of obtaining a dynamic data structure that supports insertion and removal of elements. Finally, it would be interesting to devise I/O efficient variants that are able to compete with M-trees in secondary memory. We have sketched possible alternatives but a deeper study is necessary.

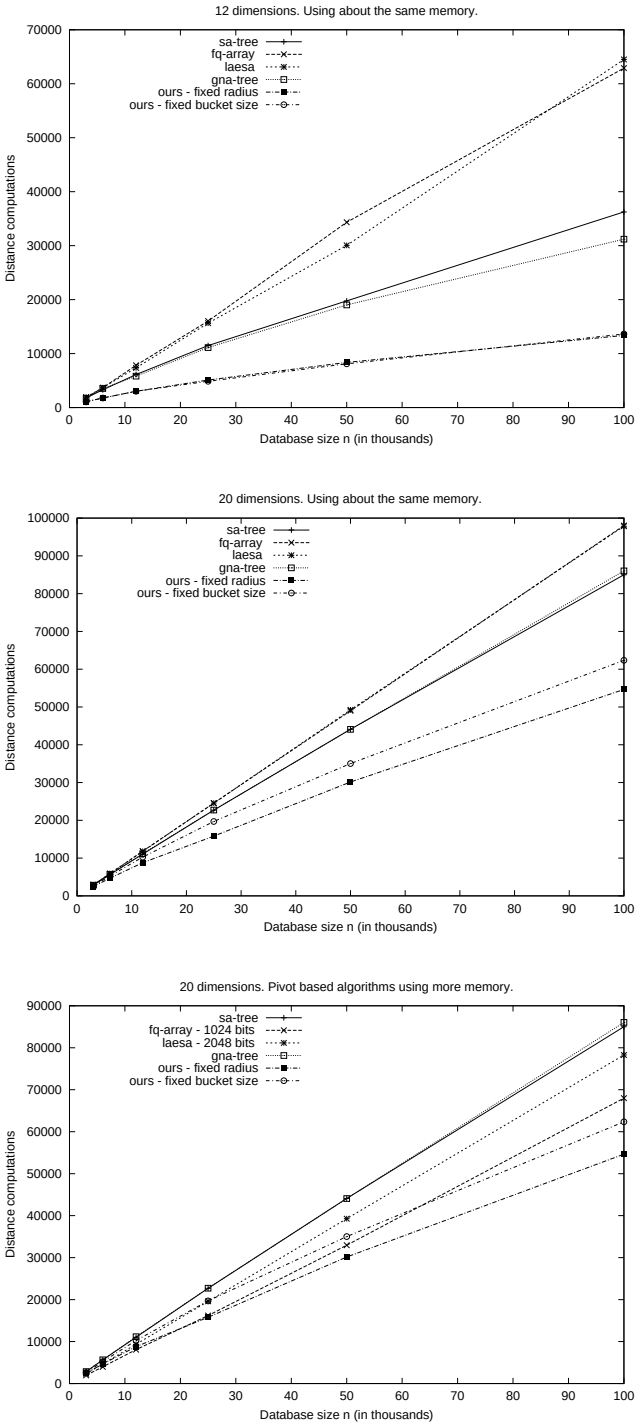

Figure 11. Comparison with existing approaches, for increasing n on fixed dimension 12 (first plot) and 20 (last two plots). In the first two plots all the algorithms use about the same memory, while the last one permits pivot based algorithms to use more memory. The query captures 0.01% of the set.

References

[1] F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3), 1991.

[2] R. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-queries trees. In *Proc. CPM'94*, LNCS 807, pages 198–212, 1994.

[3] J. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. on Software Engineering*, 5(4):333–340, 1979.

[4] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *Proc. SIGMOD'97*, pages 357–368, 1997. Sigmod Record 26(2).

[5] S. Brin. Near neighbor search in large metric spaces. In *Proc. VLDB'95*, pages 574–584, 1995.

[6] W. Burkhard and R. Keller. Some approaches to best-match file searching. *CACM*, 16(4):230–236, 1973.

[7] E. Chávez and J. Marroquín. Proximity queries in metric spaces. In *Proc. WSP'97*, pages 21–36. Carleton University Press, 1997.

[8] E. Chávez, J. Marroquín, and R. Baeza-Yates. Spaghettis: an array based algorithm for similarity queries in metric spaces. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, pages 38–46. IEEE CS Press, 1999.

[9] E. Chávez, J. Marroquín, and G. Navarro. Overcoming the curse of dimensionality. In *Proc. European Workshop on Content-Based Multimedia Indexing (CBMI'99)*, pages 57–64, 1999.

[10] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. Technical Report TR/DCC-99-3, Dept. of Computer Science, Univ. of Chile, 1999. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/survmetric.ps.gz`.

[11] 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.

[12] F. Dehne and H. Nolteimer. Voronoi trees and clustering problems. *Information Systems*, 12(2):171–175, 1987.

[13] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. ACM SIGMOD'84*, pages 47–57, 1984.

[14] I. Kalantari and G. McDonald. A data structure and an algorithm for the nearest point problem. *IEEE Transactions on Software Engineering*, 9(5), 1983.

[15] L. Micó, J. Oncina, and R. Carrasco. A fast branch and bound nearest neighbor classifier in metric spaces. *Patt. Recog. Lett.*, 17:731–739, 1996.

[16] 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. *Patt. Recog. Lett.*, 15:9–17, 1994.

[17] G. Navarro. Searching in metric spaces by spatial approximation. In *Proc. 6th South American Symposium on String Processing and Information Retrieval (SPIRE'99)*, pages 141–148. IEEE CS Press, 1999.

[18] S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. *IEEE Trans. PAMI*, 19(9):989–1003, 1997.

[19] H. Nolteimer, K. Verbarg, and C. Zirkelbach. Monotonous Bisector* Trees – a tool for efficient partitioning of complex schenes of geometric objects. In *Data Structures and Efficient Algorithms*, LNCS 594, pages 186–203. Springer-Verlag, 1992.

[20] J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *IPL*, 40:175–179, 1991.

[21] E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Patt. Recog. Lett.*, 4:145–157, 1986.

[22] P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. SODA'93*, pages 311–321, 1993.