Overcoming the Curse of Dimensionality [⋆]

Edgar Chávez[1], José L. Marroquín[2], and Gonzalo Navarro[3]

[1] Univ. Michoacana, Morelia, Mich. México. elchavez@zeus.ccu.umich.mx.
[2] Cent. de Inv. en Mat. (CIMAT), Guanajuato, México. jlm@fractal.cimat.mx.
[3] Dept. of Computer Science, Univ. of Chile, Santiago, Chile. gnavarro@dcc.uchile.cl.

Abstract. We study the behavior of pivot-based algorithms for similarity searching in metric spaces. We show that they are effective tools for intrinsically high-dimensional spaces, and that their performance is basically dependent on the number of pivots used and the precision used to store the distances. In this paper we give a simple yet effective recipe for practitioners seeking for a black-box method to plug in their applications. Besides, we introduce a new indexing algorithm that gives the minimum overall CPU search time for a given amount of memory, compared with other state-of-the-art approaches.

1 Introduction

"Proximity" or "similarity" searching is the problem of looking for objects in a set close enough to a query under a certain (expensive to compute) distance. The goal is to preprocess the set in order to minimize the number of distance evaluations at query time. This has applications in multimedia databases, machine learning, data compression, text retrieval, computational biology and function prediction, to name a few areas. A very common case arises when the objects are points in a k-dimensional Euclidean space, and well known solutions exist, such as Voronoi diagrams [1], kd-trees [5] and R-trees [12]. However, this is not the general case, and in many applications the distance is simply a metric (i.e. it satisfies the triangular inequality).

In this paper we are interested in the case of general metric spaces, where there are essentially two design approaches. One approach is based on the concept of the Voronoi [1] diagram, a data structure proven to be useful in low dimensional vector spaces. The other approach, much more popular, is based essentially in mapping the metric space onto a k-dimensional space. This last approach, the focus of this paper, leads to a family called *pivot-based* indexing algorithms, which are effective tools to handle intrinsically high-dimensional spaces (a concept explained later in this paper). This family is easily characterized with a few number of parameters governing the overall performance. We show that a particular algorithm of this family may become a reasonable choice for practitioners looking for a simple and efficient solution for similarity queries in metric spaces.

In this paper we discuss and give simple rules, i.e. recipes, to manage pivot-based algorithms. Similarity searching is a healthy and on-the-run branch of computer science, seeking for a black-box to put in applications. Our major contribution is aimed at this goal, via the discovery and systematization of a set of parameters of the pivot-based algorithms. We study two essential dimensions in the pivot-based approach, namely the number of pivots used and the discretization scheme. Both issues are thoroughly discussed through the paper.

We also introduce a new method called Fixed Queries Array (or FQA), to reduce the overall CPU search time, independently of the number of distance evaluations performed.

2 Basic Concepts

2.1 Formal Definitions

Proximity/similarity queries can be formalized using the metric space model, where a distance function $d(x, y)$ is defined for every point in a set $\mathbb{X}$. The distance function d has *metric* properties, i.e. it satisfies $d(x, y) \geq 0$ (positiveness), $d(x, y) = d(y, x)$ (symmetry), $d(x, y) = 0$ iff $x = y$ (strict positiveness), and the property allowing the existence of solutions better than brute-force for similarity queries: $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality).

The database is a set $\mathbb{U} \subseteq \mathbb{X}$, and we define the query element as q, an arbitrary element of $\mathbb{X}$. A similarity query involves additional information, besides q, and can be of two basic types:

[⋆] Supported in by CYTED VII.13 AMYRI project, and also by CONACyT grant R-28923A (first author), CONACyT (second author) and by Fondecyt grant 99-0627 (third author).

(a) Retrieve all elements which are within distance r to q, i.e. $(q,r)_d = \{u \in \mathbb{U} \ : d(q,u) \leq r\}$.
(b) Retrieve the closest elements to q in $\mathbb{U}$, i.e. $nn(q)_d = \{u \in \mathbb{U} \ : \ \forall v \in \mathbb{U},\ d(q,u) \leq d(q,v)\}$.

In this paper we are devoted to type (a) or range queries. Nearest neighbor searching, or type (b) queries, can be embedded into range queries using a branch and bound heuristic; although several dedicated algorithms have been published[11, 13, 20].

We present also the Minkowski distances for k-dimensional spaces. Under the L_p distance (r =1, 2, ... ∞), the distance between x and y is $L_p = (\sum_{i=1..k} |x_i - y_i|^p)^{1/p}$, where some particular cases are $p = 1$ (Manhattan distance), $p = 2$ (Euclidean distance) and $p = \infty$ (maximum distance). This last one deserves an explicit formula: $L_\infty = \max_{i=1..k} |x_i - y_i|$.

Finally, we discuss the issue of **intrinsic dimensionality**. The analysis of many similarity searching algorithms reveals that sub linear (and even logarithmic) average time complexity can be obtained for similarity queries. There is, however, an exponential dependence on the dimension in the case of vector spaces. The reason lies in the shape of the histogram of distances between objects. In high dimensional spaces, the histogram is more concentrated and the average distance is higher. This makes the search problem more difficult since for the same search radius more points are captured in a generalized sphere shell, e.g. the region between two d-balls centered at a point. For pivot based algorithms this sphere shell is central to discard points, as we explain later on the paper.

2.2 Related Work

Historically, the similarity searching problem appeared in the more restricted form of vector spaces, where the objects are points in a k-dimensional space (with Euclidean or Minkowski distances). General metric space algorithms inherited two major trends, very successful for vector spaces. Those models are derived from Voronoi [1] diagrams and from kd-trees [5]. We briefly discuss the first idea and then focus on pivot-based algorithms. Due to space limitation we merely sketch the approaches for similarity searching, for a more detailed discussion the reader can see [10].

Voronoi-like Algorithms The Voronoi diagram [1], or proximity graph, has been used for proximity queries in vector spaces. It is a fundamental structure in computational geometry, for solving closest point problems. It is really challenging to generalize it to metric spaces, because the algorithms to build it depend heavily on coordinate information. Nevertheless, the concept itself has inspired several approaches constructing a more or less fine approximation to either the Voronoi graph or its dual, the Delaunay triangulation. In this line we can find generalized hyper planes [17], the GNATS (Geometric Neighbor Access Trees) [7], and more recently the SB algorithm [11] and the SAT (Spatial Approximation Tree) [14]. The key idea in all these algorithms is to build a proximity graph allowing to search by approaching spatially to the query, as opposed to the pivot-based algorithms below.

Pivot-Based Algorithms The kd-trees perform a hierarchical binary decomposition of the vector space. At each level the left and right branches account for points at the left or right of a threshold in a particular coordinate. The coordinates alternate at each level.

For general metric spaces the absence of coordinates urged the design of alternative rules for space decomposition, object location and cell discarding. An entire family of algorithms are direct descendants of the kd-tree structure. Instead of using the coordinates directly, these algorithms use the distance to a set of distinguished database objects called *keys*, *vantage points* or *pivots* in the papers. Most of the schemes are tree-based data structures defining a hierarchical decomposition where the space cells coincide with leaves in the tree. Each branch, at each level, is related to the distance to some (set of) pivot(s). Subtle differences in how the pivots are selected yield to large performance differences.

If the distance function is discrete we can directly assign one branch for each different distance value. Selecting one pivot at the root level and a *different* pivot in each child node gives us the Burkhard-Keller tree (BKT) [8]. Selecting *more* than one pivot at each node is also possible and is used in [16]. Other interesting alternative is to use one pivot in each *tree level* instead of each node. This scheme is used in the Fixed Queries Trees (FQT) [3], which save distance computations in the backtracking at the expense of somewhat taller trees. Since the pivot does not reside in the nodes one can think in a further refinement of FQT, namely to arbitrarily increase the number of pivots, or equivalently the height of the tree. This arbitrarily tall trees are the Fixed Height Fixed Queries Trees (FHFQT) [2] and are proved to be more efficient than its predecessors.

If, on the other hand, the distance function is continuous, then additional work has to be done. It is impossible to assign directly one branch for each distance outcome, hence some discretization has to be carried out. In the Metric Trees [17] it is suggested to binarize the distance outcome using as threshold the median of the distance from the pivot to all its associated elements. A more complete work on the same idea is presented in the Vantage Point Trees (VPT) [19]. This tree is generalized to use more than one pivot per node and using arbitrary quantiles instead of just the median in the Multi-Vantage Point Trees (MVP) [6]. Another generalization of the same idea is to use a forest instead of a tree [20] to eliminate backtracking in limited-radius nearest neighbor search in high dimensions.

There is a trend of algorithms based simply in the use of k pivots, with little or no search structure. For each database element x, its distance to the k pivots $(d(x, p_1)...d(x, p_k))$ is stored. Given the query q, its distance to the k pivots is computed $(d(q, p_1)...d(q, p_k))$. Now, if, for some pivot p_i it holds that $|d(q, p_i) - d(a, p_i)| > r$, then we know by the triangular inequality that $d(q, a) > r$ and therefore there is no need to explicitly evaluate $d(a, p)$. All the other elements that cannot be eliminated using this rule are directly compared against the query. Notice that this is no more than a mapping of the original space onto a k-dimensional space with the L_∞ distance. Algorithms such as AESA [18], LAESA [13], and [15] are variants of this idea.

It is worth noting that all the tree-based schemes mentioned are also variants of this idea, except that they also add a data structure to avoid a linear CPU time (i.e. a linear traversal over the set).

3 A Simple Recipe: Mapping to R^k

Pivot-based algorithms can be viewed as a contractive mapping from the original metric space to a discrete k-dimensional vector space with the L_∞ distance.

The key factor is how close we can make this approximation. Adding more pivots monotonically increases the quality of the approximation. We formally state this property as a theorem.

Theorem 1. *Let $\{p_i\} \subset \{p_{i+1}\}$ be a sequence of sets of elements of the database, N the size of the database, v an arbitrary database element, q a query. Let $D_k(q, v) = \max_{1 \le j \le k}\{|d(p_j, q) - d(p_j, v)|\}$. The following chain of inequalities hold $D_i(q, v) \le D_{i+1}(q, v)$ in particular $D_N(q, v) = d(q, v)$.*

Proof. Since the set of pivots form a chain of contentions, as i increases the maximum cannot decrease. For the last assertion, for D_N we have already used all of the pivots (i.e. compared with every database element), and by the triangle inequality $d(v, q) \ge d(p_j, q) - d(p_j, v)$ for any p_j, with equality when $p_j = v$.

A simple lesson is learned from the above theorem: one can increase the performance of a pivot-based algorithm by adding more pivots. Nevertheless, this implies using more memory each time we add a new pivot. If we simply use an array to store the distances, then we have to perform a linear pass over this array to compute the L_∞-ball or candidate list. A tree-based data structure allows, on the other hand, to build this L_∞-ball in sub linear time.

An array uses a small amount of memory using a linear pass to isolate the set of candidate points, while a tree speeds up the process but uses a large amount of memory to maintain the index. It seems to be a two-fold alternative. We present now an approach reaching both: a speed up, and small amount of memory.

3.1 Internal and External Complexity

In [10] a model for effectively describing the complexity of an indexing algorithm is presented. A common measure of time complexity for similarity indexing is the number of distance computations, that are by far the most complex operation in the process. According to [10] one can define the *internal complexity* as the number of distance computations needed to isolate a list of candidates to satisfy the query. The candidate list is trimmed to obtain the outcome, and the number of objects in the candidate list is the *external complexity* (because one has to examine them to obtain the query outcome).

For pivot-based algorithms the internal complexity is the number of pivots, while the external complexity is the number of database points not filtered by the pivots. One can increase the internal complexity according to Theorem 1 knowing that the external complexity must decrease (or at least, not increase).

3.2 A Unified View for Pivot-Based Algorithms

We can consider the pivot-based algorithms as having two parts: An array with the coordinates of each point, and a data structure to isolate a list of candidate points using the information of the array. In some realizations of this idea, the array is stored partially in the final data structure, to allow the use of more pivots. It will be lengthly to describe here this unified view, we refer the reader to [10] for a more detailed discussion of these ideas.

In the preceding section we have shown that increasing the number of pivots, i.e. the dimension of the target vector space, is effective to overcome the so called curse of dimensionality. On the other hand, we saw that the space requirements are increased. The question is, then, given an amount of available memory, how well can we use it in order to have more pivots.

Some approaches such as the FHFQT [2] try to reduce the side computations with a tree data structure. This structure, however, takes a lot of space and the net effect is that less pivots can be used in practice. Other approaches use little or no extra space (apart from the distances from each element of the database to the k pivots selected). These are [15, 13, 9], some of which also present search algorithms trying to reduce the CPU time.

4 Fixed Queries Arrays

For a traditional (exact) search problem, one can select between an array and a tree to implement essentially the same idea: *binary searching.* Both implementations have the same theoretical complexity for searching; but a tree have facilities to dynamically insert/delete points. Our intention is to build a scheme where the array itself can be used for similarity searching directly, without an indexing structure and using a sublinear amount of time.

In this section we introduce the Fixed Queries Array (FQA). This is a simple data structure which also allows sub linear CPU search time. Given k pivots, its performance is exactly the same as the methods just mentioned, but the search strategy is different: the most interesting feature of FQAs is their ability to reduce the *precision* of the distances stored, exchanging it for more pivots.

4.1 The FQA Structure

First assume that the set of possible distances is discrete. Given each element of the database, a list of its distances to the k pivots is stored. In the FQA, this list is considered as a sequence of k integers. The structure simply stores the database elements lexicographically sorted by this sequence of distances, that is, the elements are first sorted by their distance to the first pivot, those at the same distance to the first pivot are sorted by their distance to the second pivot, and so on. As more and more keys are added, the array becomes more and more "sorted".

The result has strong relations to the FHFQT. If the leaves of the tree are traversed in order, the outcome is precisely the order imposed in the FQA. Moreover, the search algorithm of the FHFQT is inherited by the FQA. Each node of the FHFQT corresponds to a range of cells in the FQA. If a node descends from another in the tree, its range is a subrange of the other in the array[1]. Hence, each time the tree algorithm moves from a node to a child in the tree, we mimic the movement in the array, by binary searching the new range inside the current one. This binary search does not perform extra distance evaluations. The net result is that the number of distance evaluations is the same, and the CPU complexity is multiplied by an additional $O(\log n)$ factor. As proved in [4], the FHFQT has $O(n^\alpha)$ CPU complexity $(0 < \alpha < 1)$, and therefore the FQA is $O(n^\alpha \log n)$. The number of distance evaluations can be made $O(\log n)$ by using $\Theta(\log n)$ pivots.

To make the idea more clear, we show explicitly the search algorithm. Given a query q to be searched with tolerance r and k pivots $p_1 ... p_k$, we measure $d_1 = d(q, p_1)$. Now, for every i in the range $d_1 - r$ to $d_1 + r$, we binary search in the array the range where the first coordinate is i. Once that range is computed, for each i, we recursively continue the search on the sub array found, from the pivot p_2 on. This is equivalent to recursively entering into the i-th subtree of the FHFQT. The search finishes when we used the k pivots, and at that point the remaining sub arrays are sequentially checked. The recursive procedure obviously finishes also when the remaining sub array is empty.

[1] This has close resemblances to suffix trees and suffix arrays, two text retrieval data structures.

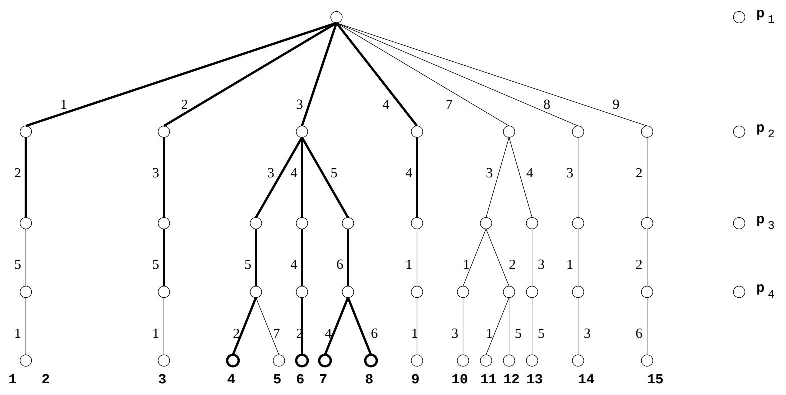

Fig. 1. A tree implementation (FQA) for a small example

4.2 An Example

Consider the FHFQT of Figure 1. Each branch from the root represents a distance to pivot p_1. Branches from the second-level nodes refer to the distances to p_2, and so on. Given a query $(q, r)_d$, the search algorithm enter, at level i in the tree, only those branches within the interesting interval $d(q, p_i) \pm r$. Consider $r = 2$ and $\{d(q, p_i)\} = \{3, 4, 5, 4\}$. Branches labeled $[1, 2, 3, 4]$ in the first level will be examined and, recursively, all branches below them will be traversed according to the appropriate interval for their respective levels. When a branch is outside the interesting interval it is pruned, e.g. branches $[7, 8, 9]$ in the example. At the end, database elements $\{4, 6, 7, 8\}$ will remain in the candidate list, and will be tested against the query to see if they should be in the query outcome. The memory usage of this tree is 215 bytes: 45 nodes assuming 5 bytes per node (a very efficient implementation).

The equivalent FQA stores the elements in the left-to-right order shown in Figure 1, keeping the four distances for each element. Figure 2 illustrates the search process.

We have four pivots, and each row in the four tables a,b,c and d represents a branch in the tree; these in turn represent the distances from the database point to the appropriate pivot. For a query q we compute the vector $(d(q, p_0), \cdots, d(q, p_4))$, in this case $(3, 4, 5, 4)$. The search radius is 2. We have to search the intervals $(\{1, 5\}, \{2, 6\}, \{3, 7\}, \{2, 6\})$ respectively. Figure 2 illustrates this. With binary search we find the intervals in the first column (boldface rows). In each one of the four tables, we show in boldface the candidates after each search step. Table (a) is equivalent to the first level in the tree, and so on for the rest of them. We can easily check that binary searching intervals in each column is equivalent to bounding the search in the appropriate levels in the tree.

It is worth to observe that lexicographical ordering allows one to use binary searching in subsequent columns. Consider for example rows beginning with a 3: all the elements of the second column are also sorted in increasing order, and so on.

	(a)			
	1	**1**	**4**	**4**
	1	**2**	**5**	**1**
	1	**2**	**5**	**1**
	2	**3**	**5**	**1**
	3	**3**	**5**	**2**
	3	**3**	**5**	**7**
	3	**4**	**4**	**2**
(1,5)	**3**	**5**	**6**	**4**
	3	**5**	**6**	**6**
	4	**4**	**1**	**1**
	7	3	1	3
	7	3	2	1
	7	3	2	5
	7	4	3	5
	8	3	1	3
	9	2	2	6

	(b)			
	1	**1**	4	4
	1	**2**	**5**	**1**
	1	**2**	**5**	**1**
	2	**3**	**5**	**1**
	3	**3**	**5**	**2**
	3	**3**	**5**	**7**
	3	**4**	**4**	**2**
(2,6)	**3**	**5**	**6**	**4**
	3	**5**	**6**	**6**
	4	**4**	1	1
	7	3	1	3
	7	3	2	1
	7	3	2	5
	7	4	3	5
	8	3	1	3
	9	2	2	6

	(c)			
	1	**1**	4	4
	1	**2**	**5**	1
	1	**2**	**5**	1
	2	**3**	**5**	1
	3	**3**	**5**	**2**
	3	**3**	**5**	**7**
	3	**4**	**4**	**2**
(3,7)	**3**	**5**	**6**	**4**
	3	**5**	**6**	**6**
	4	**4**	1	1
	7	3	1	3
	7	3	2	1
	7	3	2	5
	7	4	3	5
	8	3	1	3
	9	2	2	6

	(d)			
	1	**1**	4	4
	1	**2**	**5**	1
	1	**2**	**5**	1
	2	**3**	**5**	1
	3	**3**	**5**	**2**
	3	**3**	**5**	7
	3	**4**	**4**	**2**
(2,6)	**3**	**5**	**6**	**4**
	3	**5**	**6**	**6**
	4	**4**	1	1
	7	3	1	3
	7	3	2	1
	7	3	2	5
	7	4	3	5
	8	3	1	3
	9	2	2	6

Fig. 2. Searching an FQA.

It is clear that the candidate list using either representation is unchanged. In the array based search we have to pay $O(\log n)$ (the cost of a binary search) to simulate a visit to a branch. If we visit m nodes in the tree, we use $O(m \log n)$ time in the array.

5 The Continuos Case

We assumed that the distance is discrete, and this is not the general case. Observe that the FHFQT and the FQA do not work well if the distance is continuous. Hence, it is necessary to define ranges in the continuum of possible outcomes of the distance function and assign them to a small set of discrete values. This idea, however, has its own value, as we need less space to store these discretized values.

In general, instead of storing k coordinates separately, we consider the whole sequence of (discretized) values as an unsigned b bits integer. Each group of b_s bits represents the distance from the database element to a pivot, i.e. we can represent 2^{b_s} values. Since the most significant bits are assigned to the first pivots, we have the lexicographical ordering inherited by the integer ordering of the b bits. Hence, we can have more pivots at the expense of storing less bits for the distances. This allows an extra degree of freedom in the use of the available memory.

It is worth noting that the representation is not tightly linked with the discretization rule. One can use any suitable rule to assign database points to branches in the tree. The next section is devoted to the quest for the optimal scheme of discretization.

5.1 Discretizing Schemas

In this section we discuss two discretizing schemes for the Fixed Queries Arrays.

FHQA For each pivot, independently, we obtain $Dmax = \max\{d(p_i, u)\}$ and $Dmin = \min\{d(p_i, u)\}$ for $u \in \mathbb{U}$, and $u \neq p_i$. The range $Dmax - Dmin$ is divided then in 2^{b_s} parts, and each binary number x will be associated to the interval $[Dmin + x\ (Dmax - Dmin)/2^{b_s}, Dmin + (x+1)\ (Dmax - Dmin)/2^{b_s})$.

We call this scheme *fixed slices*, and the resulting FQA is called *FHQA*.

FMVPA In the above scheme we have no control on the number of database points falling in a particular interval. It is even possible to have *empty* slices, where no database point falls. This motivates another discretization scheme where the control variable is the number of database points falling in a given interval.

The procedure ensures that in each interval there are exactly $n/2^{b_s}$ points. In other words, we find 2^{b_s} quantiles. We call this scheme *fixed quantiles*, and the resulting FQA is called *FMVPA*.

6 Some Experiments

We have selected a sample of 100,000 uniformly distributed real vectors on the unit cube for our experiments and used the L_2 (Euclidean) distance. Although this is a space with coordinates, we treat it as a general metric space (not making use of the coordinates). This allows us to control precisely the effective dimension of the data. All the graphs show how many distance computations are needed to satisfy a query retrieving 0.01% of the database.

Note that for a fixed amount of memory there are many possible combinations of pivots and resolution for both FHMVPA and FHQA. For example if we have 32 bits for each database point, then we can choose to have 32 1-bit pivots, or 16 2-bit pivots, or 8 4-bit pivots, etc. As it is not clear what is the best combination, we try to figure out it and obtain a recommendation for practitioners.

Another unclear issue is what is the best scheme for FQA: FHMVPA or FHQA. In the graphs the schemes are named "FHMVPA/FHQA $h - b$", where h is the number of pivots used and b is the number of bits per pivot. If the same memory is used then $h2^b$ is constant.

6.1 Memory Optimization

FHQA In Figure 3 (top) we observe that for a small amount of memory, the difference between schemes is negligible. However, using 256 bits (8 words), the best is 4 bits per pivot in almost every dimension.

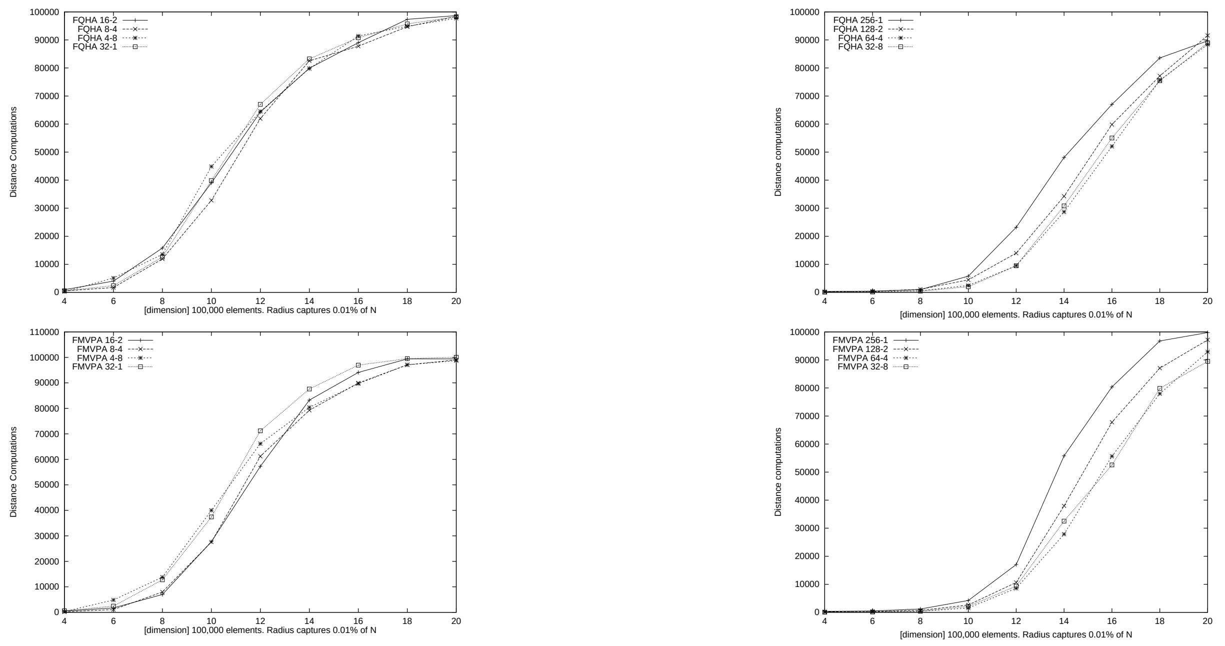

Fig. 3. FHQA (top) and FMVPA (bottom) using 32 bits (left) and 256 bits (right), for several combinations of resolution/pivots for a fixed amount of memory.

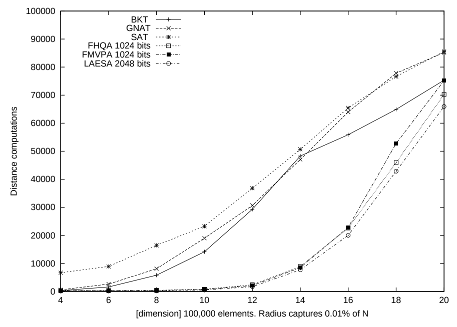

Fig. 4. Comparing FQA with state-of-the-art approaches.

FHMVPA Figure 3 (bottom) shows the same behavior for the FHMVPA, the optimal selection being 4 bits per pivot. However, the slopes in the curves are different. It is clear that the FHQA is the best selection as a discretization scheme in this experiment.

6.2 Against-all Comparison

In Figure 4 we show plots of the number of distance computations vs. dimension for state of the art approaches. The parameters of the algorithms were chosen empirically and following the original suggestions of their authors. The memory usage can be controlled explicitly for the pivot-based algorithms. Observe how one can decrease arbitrarily the number of distance computations at the expense of more memory.

Figure 4 shows many interesting facts on similarity searching. First observe how the performance of all the algorithms decreases as the dimension increases. Some algorithms degrade faster than others. A simple extrapolation of the behavior allows one to predict that as the dimension increases no algorithm will be able to save a single distance computation. This is the experimental face of the "curse of dimensionality".

The other important conclusion is that pivot based algorithms can be tuned to beat any algorithm, providing enough memory for them. One of the least demanding pivot based scheme is LAESA, which does not discretize and stores a plain array with the k coordinates. From the experiment we see that

LAESA needs twice as many bits as our FQA to match the same complexity. Moreover, FQA has sublinear side computations ($O(n^\alpha \log n)$) while LAESA is linear ($O(n)$ to $O(kn)$). Other sublinear-time schemes such as FHFQT need much more space to reach the same number of computations as our FQA.

7 Conclusions and Future Work

We have presented a study of the behavior of pivot-based algorithms for similarity searching in metric spaces. We have shown that pivot-based algorithms are effective tools to deal with intrinsically high-dimensional spaces. We have also shown that their performance is basically dependent on the number of pivots used. We have presented a new data structure, called Fixed Queries Array (FQA), to reduce the number of CPU computations and the space requirements of the index.

The FQA is shown to be a simple yet effective structure for this problem. Its most important feature is that it allows effective space usage, not only because it puts minimal overhead over the storage requirements but also because it allows to reduce the precision with which the coordinates are stored. This reduced precision is exchanged for more pivots.

We will guide our research trough the quest for less expensive (in memory usage) alternatives to overcome the curse of dimensionality.

References

1. F. Aurenhammer. Voronoi diagrams – a survey of a fundamental geometric data structure. *ACM Computing Surveys*, 23(3), 1991.
2. R. Baeza-Yates. Searching: An algorithmic tour. In Allen Kent and James G. Willias, editors, *Encyclopedia of Computer Science and Technology*, volume 37, pages 331–359. Marcel Dekker, Inc., 1997.
3. R. Baeza-Yates, W. Cunto, Udi Manber, and Sun Wu. Proximity matching using fixed-queries trees. In M. Crochemore and D. Gusfield, editors, *5th Combinatorial Pattern Matching*, LNCS 807, pages 198–212, Asilomar, CA., June 1994. Springer-Verlag.
4. R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proc. String Processing and Information Retrieval (SPIRE'98)*, pages 14–22. IEEE CS Press, 1998.
5. J. Bentley. Multidimensional binary search trees in database applications. *IEEE Trans. on Software Engineering*, 5(4):333–340, 1979.
6. T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. Manuscript.
7. S. Brin. Near neighbor search in large metric spaces. In *Proc. VLDB'95*, pages 574–584, 1995.
8. W. A. Burkhard and R.M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, April 1973. The source for BK-trees.
9. 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)*, Cancun, Mexico, September 1999. To appear. `ftp://garota.fismat.umich.mx/pub/users/elchavez/spa.ps.gz`.
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. Submitted. `ftp://ftp.dcc.uchile.cl/-pub/users/gnavarro/survmetric.ps.gz`.
11. K. L. Clarkson. Nearest neighbor searching in metric spaces: Some experimental results. 1999.
12. A. Guttman. R-trees: a dynamic index structure for spatial searching. In *The ACM SIGMOD International Conference on the management of data*, pages 47–57, 1984.
13. L. Micó, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.
14. G. Navarro. Searching in metric spaces by spatial approximation. In *Proc. String Processing and Information Retrieval (SPIRE'99)*, Cancun, Mexico, September 1999. To appear. `ftp://ftp.dcc.uchile.cl/pub/users/-gnavarro/metric.ps.gz`.
15. S. Nene and S. Nayar. A simple algorithm for nearest neighbor search in high dimensions. Technical Report CUCS-030-95, Dept. of Computer Science, Columbia University, NY, October 1995.
16. M. Shapiro. The choice of reference points in best-match file searching. *Comm. ACM*, 20(5):339–343, 1977.
17. J. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.
18. E. Vidal. An algorithm for finding nearest neighbors in (approximately) constant average time. *Pattern Recognition Letters*, 4:145–157, 1986.
19. P. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proc. ACM-SIAM SODA'93*, pages 311–321, 1993.
20. Peter N. Yianilos. Excluded middle vantage point forests for nearest neighbor search. Technical report, NEC Research Institute, Princeton, NJ, July 1998.