Fixed Queries Array: A Fast and Economical Data Structure for Proximity Searching *

Edgar Chávez (elchavez@zeus.ccu.umich.mx)
Univ. Michoacana, Morelia, Mich. México.

José L. Marroquín (jlm@cimat.mx)
Cent. de Inv. en Mat. (CIMAT), Guanajuato, México.

Gonzalo Navarro (gnavarro@dcc.uchile.cl)
Dept. of Computer Science, Univ. of Chile, Santiago, Chile.

Abstract. Pivot-based algorithms are effective tools for proximity searching in metric spaces. They allow trading space overhead for number of distance evaluations performed at query time. With additional search structures (that pose extra space overhead) they can also reduce the amount of side computations. We introduce a new data structure, the Fixed Queries Array (FQA), whose novelties are (1) it permits sublinear extra CPU time without any extra data structure; (2) it permits trading number of pivots for their precision so as to make better use of the available memory. We show experimentally that the FQA is an efficient tool to search in metric spaces and that it compares favorably against other state of the art approaches. Its simplicity converts it into a simple yet effective tool for practitioners seeking for a black-box method to plug in their applications.

Keywords: Metric spaces, similarity search, range search, fixed queries tree.

1. Introduction

Proximity 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 is a very active branch of computer science, seeking for a black-box to put in applications such as multimedia databases, machine learning, data compression, text retrieval, computational biology and function prediction, to name a few.

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 (Aurenhammer, 1991), kd-trees (Bentley, 1975) and R-trees (Guttman, 1984). However, this is not the general case, and in many applications the distance is simply a metric (i.e. it just satisfies the triangular inequality).

* Supported in part by CYTED VII.13 AMYRI project, and also by CONACyT grant R-28923A (first author), CONACyT (second author) and Fondecyt grant 1-000929 (third author).

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 diagram (Aurenhammer, 1991), 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.

This family has interesting properties, such as the ability to pay more space overhead (basically by incrementing k) in order to reduce the number of distance evaluations at query time. The higher the intrinsic dimension of the space (a concept that we explain later), the more pivots are needed to obtain the same performance, a phenomenon known as the "curse of dimensionality". Therefore, efficient space usage is an issue for pivot-based algorithms.

On the other hand, it is not always realistic to assume that the distance function is so expensive to compute that all the other side computations can be neglected. Therefore, many pivot-based algorithms add extra data structures (which pose more space overhead) in order to reduce the side computations. This does not reduce the number of distance evaluations, but the search is in practice faster.

In this paper we introduce a new data structure called Fixed Queries Array (or FQA), which has two interesting properties. First, it is the first data structure able of achieving a sublinear (in the database size) number of side computations *without* using any extra space. Second, it is able to trade number of pivots k for their precision, so as to optimize the usage of the available space.

We compare experimentally the FQA against other state of the art approaches and show that it is a simple and effective alternative. The FQA is a very appealing choice for practitioners looking for a simple and efficient solution for proximity queries in metric spaces.

The paper is organized as follows. In Section 2 we give the basic formal definitions and review related work. In Section 3 we present the pivot-based approach and discuss efficiency measures. In Section 4 we introduce the FQAs. Section 5 presents our experimental results. Finally, we give our concluding remarks in Section 6. A preliminary version of this work appeared in (Chávez et al., 1999b).

2. Basic Concepts

2.1. Formal Definitions

Proximity 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 proximity queries: $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality).

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

Range query: retrieve all elements which are within distance r to q, i.e. $(q, r)_d = \{u \in \mathbb{U} \ : d(q, u) \leq r\}$.

Nearest neighbor query: 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 range queries. Nearest neighbor queries can be embedded into range queries using a branch and bound heuristic; although several dedicated algorithms have been published (Clarkson, 1999; Micó et al., 1994; Yianilos, 1999).

Vector spaces are a particular case of metric spaces where the elements are k-dimensional coordinates under the L_p distance (p =1, 2, ... ∞). This is defined as follows: the L_p distance between x and y is

$$L_p((x_1, \ldots, x_k), (y_1, \ldots, y_k)) = \left(\sum_{1 \leq i \leq k} |x_i - y_i|^p \right)^{1/p} ,$$

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:

$$L_\infty((x_1, \ldots, x_k), (y_1, \ldots, y_k)) = \max_{1 \leq i \leq k} |x_i - y_i| \ .$$

Figure 1 illustrates.

The goal of a proximity search algorithm is to build in advance a data structure (called "index") so as to minimize the search cost at query time. There are three main terms in this cost, namely the number of distance computations, the extra CPU cost and the I/O cost. In this paper we concentrate in the first two, assuming that the

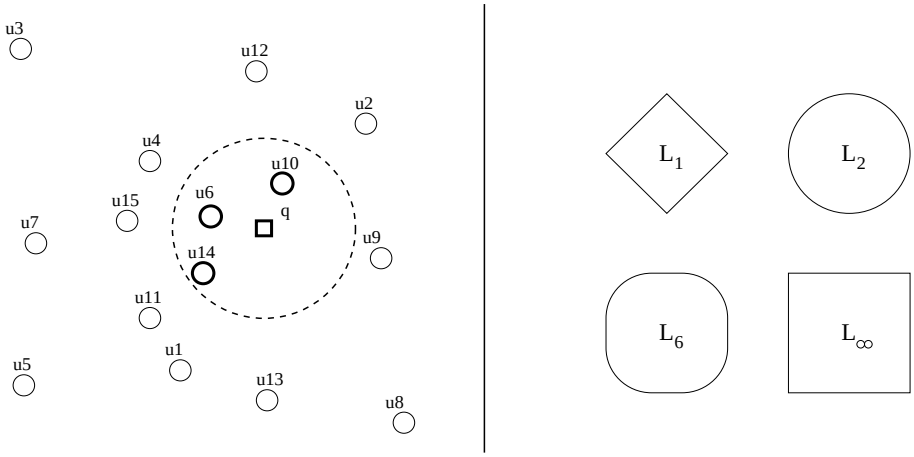


Figure 1. On the left, an example of a range query on a set of points. On the right, the set of points at the same distance to a center point, for different L_p distances.

index fits in main memory. There exist currently very few approaches to the secondary memory problem on metric spaces (see (Ciaccia et al., 1997)). On the other hand, the importance of the extra CPU cost (or "side computations") depends on the application, namely on how costly to compute is the distance function.

2.2. Related Work

Historically, the proximity searching problem appeared in the more restricted form of vector spaces, where the objects are points in a k-dimensional space (with L_p distances). General metric space algorithms inherited two major trends, very successful for vector spaces. Those models are derived from Voronoi diagrams (Aurenhammer, 1991) and from kd-trees (Bentley, 1975). We briefly discuss the first idea and then focus on pivot-based algorithms. For a more thorough discussion see (Chávez et al., 1999c).

2.2.1. *Voronoi-like Algorithms*

The Voronoi diagram (Aurenhammer, 1991), 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 hyperplanes (Kalantari and McDonald, 1983; Dehne and

Nolteimer, 1987; Uhlmann, 1991b), the GNATs (Geometric Neighbor Access Trees) (Brin, 1995), and more recently the M-trees (Ciaccia et al., 1997), the SB algorithm (Clarkson, 1999) and the SAT (Spatial Approximation Tree) (Navarro, 1999). The key idea in all these algorithms is to cluster the space so as to search by approaching spatially to the query, as opposed to the pivot-based algorithms below.

2.2.2. *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. This is combined with the triangular inequality to obtain a discarding rule similar to that of *kd*-trees.

Most of these schemes are tree-based data structures defining a hierarchical decomposition where the space cells coincide with leaves in the tree. The simplest example is the Burkhard-Keller Tree (BKT), a data structure designed for distance functions yielding discrete values. Each node of the tree corresponds to a different pivot p, and each descending branch to a distance from p. That is, all the elements at distance i from p are put in the i-th subtree of the corresponding node. The subtrees are recursively built with the same rule. At search time, for a query $(q, r)_d$, we backtrack in the tree entering only in the subtrees numbered $d(q, p) - r$ to $d(q, p) + r$, as the other elements can be discarded using the triangular inequality.

Many other variations over the same idea exist. We can select more than one pivot at each node, as in (Shapiro, 1977). Other interesting alternative is to use one pivot in each *tree level* instead of each node. This scheme is used in the Fixed Queries Tree (FQT) (Baeza-Yates et al., 1994), which saves distance computations in the backtracking at the expense of somewhat taller trees. Since the pivots do 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. These arbitrarily tall trees are the Fixed Height Fixed Queries Trees (FHFQT) (Baeza-Yates, 1997), which are experimentally shown to be more efficient than their predecessors.

If, on the other hand, the distance function is continuous, then this scheme does not work because it is impossible to assign directly

one branch for each distance outcome. Hence some *discretization* of the distances has to be carried out. In the Metric Trees and Vantage Point Trees (VPTs) (Uhlmann, 1991a; Yianilos, 1993) it is suggested to binarize the distance outcome by using as threshold the median of the distances from the pivot to all its associated elements. This guarantees that the tree is well balanced. The VP-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 Tree (MVP) (Bozkaya and Ozsoyoglu, 1997). Another generalization of the same idea is to use a forest instead of a tree (Yianilos, 1999) to eliminate backtracking in limited-radius nearest neighbor search.

A different trend of algorithms based on pivots stores the information in array form. For each database element a, its distance to the k pivots $(d(a,p_1)...d(a,p_k))$ 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. Algorithms such as AESA (Vidal, 1986) and LAESA (Micó et al., 1994) are variants of this idea. Note, however, that in this case there is no search structure to help reduce the extra CPU time. That is, despite that the number of distance computations using k pivots is the same as for a FHFQT of height k, in this case we need to traverse the array of distances element by element, for a minimum of $\Omega(n)$ extra CPU time. A few proposals to reduce the extra CPU time while keeping the array structure exist, most notably (Nene and Nayar, 1997) and the Spaghettis (SPA) (Chávez et al., 1999a), which independently sort the n distances along each coordinate in order to replace the linear traversal by binary searches of the range $d(q,p_i) \pm r$. In this case, however, they need to store additional links to be able to retrieve the permutation performed by the sorting process.

It is worth noting that all the tree and array based schemes mentioned are variants of the same idea, except that they add different (or no) data structures to avoid a linear CPU time (i.e. a linear traversal over the set). The methods differ also in the form they select the pivots, but the general principle is that, given an element a and a pivot p, if we store $d(a,p)$ somewhere in the index, then at query time we can avoid computing $d(q,a)$ whenever $|d(q,p) - d(a,p)| > r$. This is the essence of pivot based algorithms. Figure 2 illustrates.

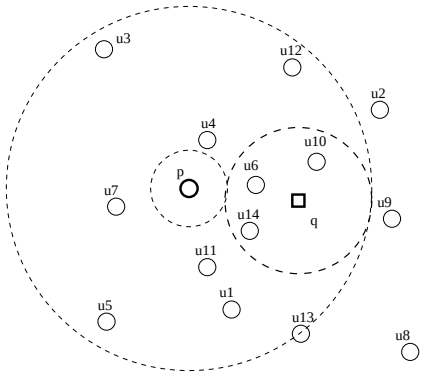


Figure 2. Using one pivot. The points between both rings centered at p qualify for the next iteration.

3. Pivot Based Algorithms as a Mapping to R^k

An abstract view of a pivot based algorithm is as follows. We select a set of k pivots $\{p_1, \ldots, p_k\}$. At indexing time, for each database element a, we compute and store $\Phi(a) = (d(a,p_1)...d(a,p_k))$. At query time, for a query $(q,r)_d$, we compute $\Phi(q) = (d(q,p_1)...d(q,p_k))$. Now, we can discard every $a \in \mathbb{U}$ such that, for some pivot p_i, $|d(q,p_i) - d(a,p_i)| > r$, or which is the same, we discard every a such that

$$\max_{1 \le i \le k} |d(q,p_i) - d(a,p_i)| \quad = \quad L_\infty(\Phi(a), \Phi(q)) \quad > \quad r \ .$$

This shows that pivot-based algorithms can be viewed as a *mapping* Φ from the original metric space $(\mathbb{X}, d)$ to a k-dimensional vector space with the L_∞ distance, namely $(\mathbb{R}^k, L_\infty)$. Moreover, this mapping is *contractive*, i.e. $L_\infty(\Phi(x), \Phi(y)) \le d(x,y)$ because of the triangle inequality.

Hence, the underlying idea of pivot based algorithms is to project the space into a new space where the distances are reduced. We search in the new space with the same radius r, which guarantees that no answer will be missed. On the other hand, elements that should not be in the answer in the original space are selected in the projected space. This is the reason why it is necessary to check directly with the d distance all the elements a that cannot be discarded in the projected space. Figure 3 illustrates.

3.1. Internal and External Complexity

The key factor is how close can we make this approximation. That is, the L_∞ distance in the projected space lower bounds d, and we would like it to be as close to d as possible. Adding more pivots (i.e. increasing

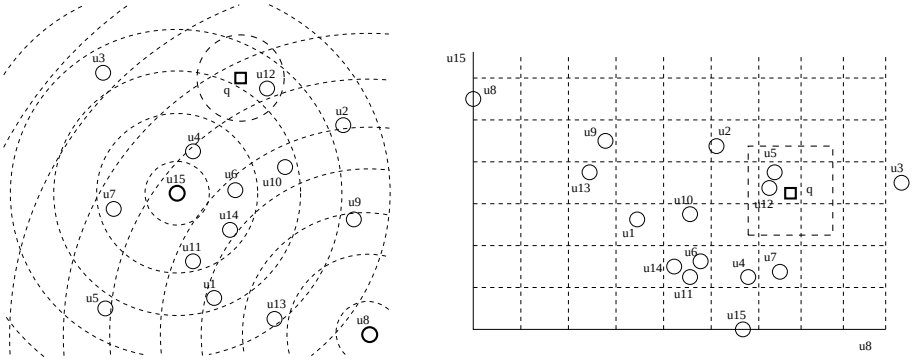


Figure 3. An equivalence relation induced by intersecting rings centered in two pivots u_8 and u_{15}, and how a query is transformed.

k) monotonically increases the quality of the approximation, since the L_∞ distance is the maximum between k distances. This is easy to prove formally:

Lemma 1. Let $\{p_i\} \subset \{p_{i+1}\} \subseteq \mathbb{U}$ be a sequence of subsets of the database, n the size of the database, a an arbitrary database element, q a query. Let $D_k(q,a) = \max_{1 \le j \le k}\{|d(p_j,q) - d(p_j,a)|\}$. The following chain of inequalities holds: $D_i(q,a) \le D_{i+1}(q,a)$, in particular $D_n(q,a) = d(q,a)$.

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 $d(a,q) \ge d(p_j,q) - d(p_j,a)$ for any p_j, with equality when $p_j = a$.

Figure 4 shows an empirical verification of this assertion. We generated random uniformly distributed vectors in the metric space $([0,1]^{32}, L_2)$ and show the histogram of distances for the original distance L_2 and for the pivot distance L_∞ obtained with $k = 16$ and $k = 512$ pivots. As can be seen, the histogram of L_∞ approximates better the original L_2 as k grows.

A simple lesson is learned from the above discussion is that one can improve the quality of the approximation of a pivot-based algorithm by adding more pivots. Nevertheless, this implies using a larger k, and we also need to perform k distance evaluations to obtain $\Phi(q)$. This leads to a clear separation of the number of distance evaluations performed at search time, in two classes.

Definition 1. Let a search algorithm be based in a mapping Φ from $(\mathbb{X}, d)$ to $(\mathbb{R}^k, L_\infty)$, and let $(q,r)_d$ be a range query. Then the *inter-*

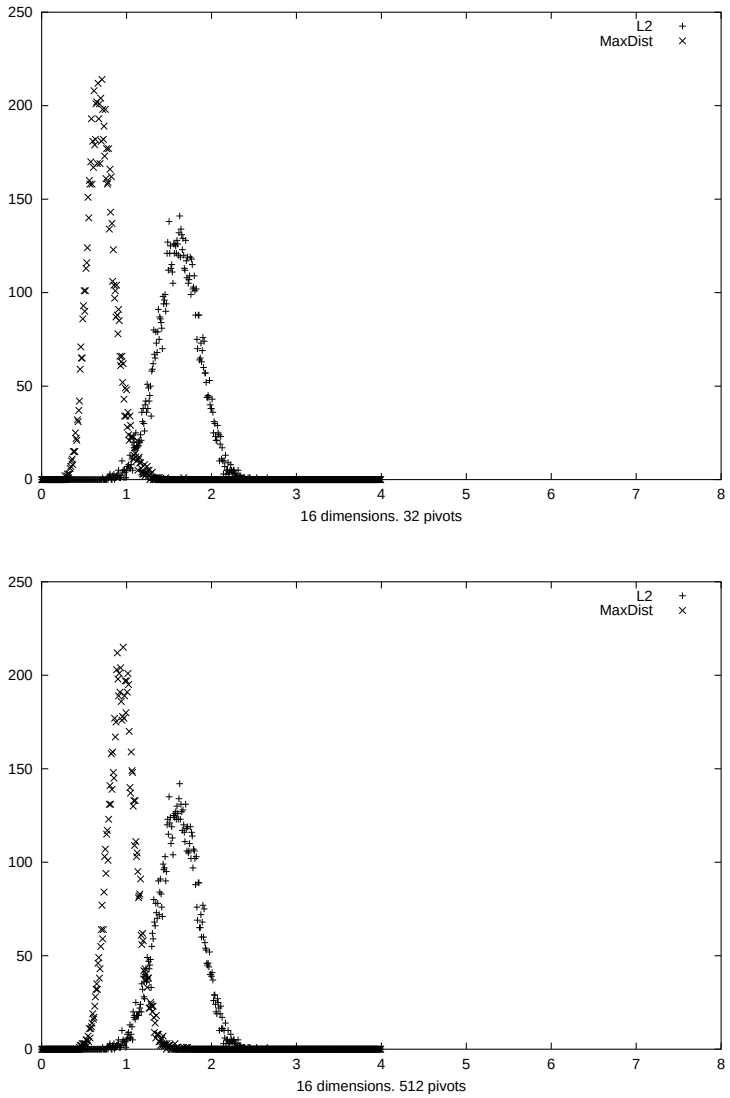


Figure 4. Histograms comparing the L_2 distance in a 16-dimensional Euclidean space and the pivot distance (MaxDist) obtained using different numbers k of pivots. On the top $k = 32$ and on the bottom $k = 512$.

nal complexity of the algorithm is k, while its *external complexity* is $|(\Phi(q), r)_{L_\infty}|$, the size of the outcome of the query in the mapped space.

That is, the internal complexity is the cost to compare q against the k pivots to obtain its k coordinates in the target space, while the external complexity is the cost to check the list of candidates remaining after filtering the database using the coordinates in the mapped space.

What Lemma 1 says is that we can decrease the external complexity by increasing the internal complexity (number of pivots). It is clear that there is an optimum k where the sum of internal plus external complexity is minimized. Figure 5 shows an experiment with random uniformly distributed vectors in $([0,1]^8, L_2)$, where we have used different number of pivots and the optimum is reached for k close to 110.

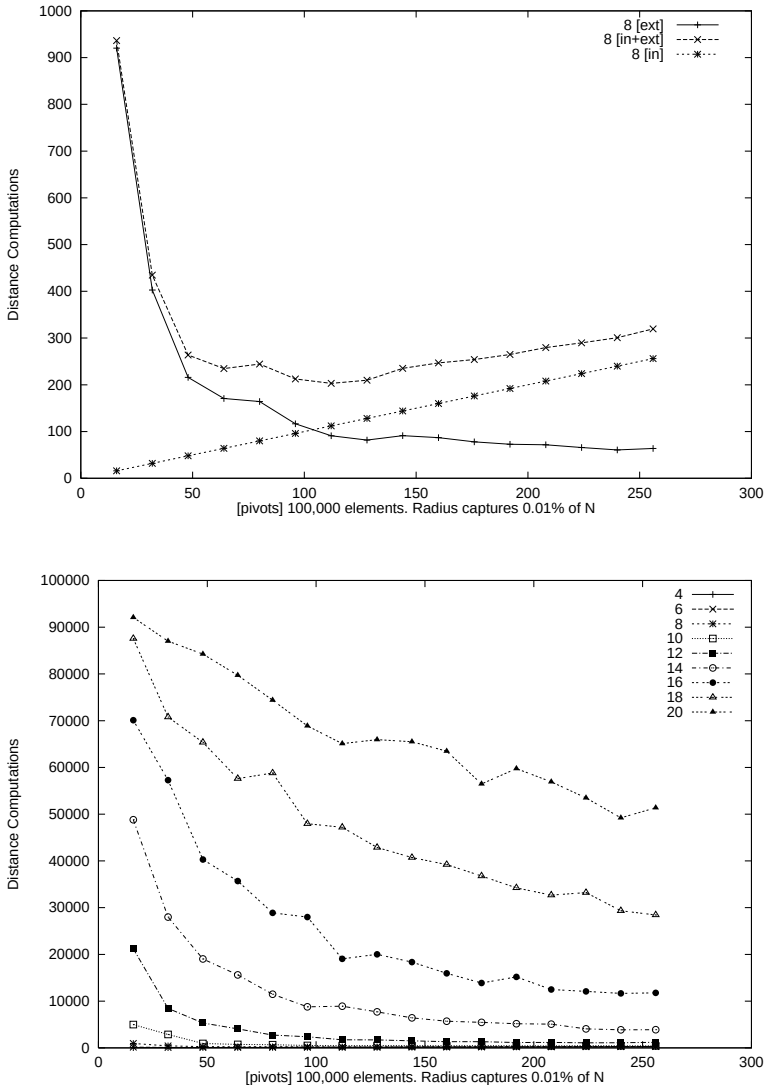


Figure 5. On the top, internal, external and overall distance evaluations in 8 dimensions, using different number of pivots k. On the bottom, overall distance evaluations for more dimensions.

3.2. Intrinsic Dimensionality

What Figure 5 also shows is that this optimum is larger as the dimension of the space grows. That is, it is convenient to use more and more pivots as the dimension grows.

In (Chávez et al., 1999c) the *intrinsic dimension* of a general metric space is defined in terms of its histogram: $\rho = \frac{\mu^2}{2\sigma^2}$, where μ and σ are the mean and standard deviation of the histogram of distances in the metric space. As for vector spaces, a more skewed histogram means a higher intrinsic dimension. Moreover, the intrinsic dimension of a random k-dimensional vector space is shown to be $\Theta(k)$. A fundamental contribution of (Chávez et al., 1999c) is to prove that (1) a lower bound on the average number of distance evaluations performed by a pivot-based algorithm, for randomly chosen pivots, is $\Theta(\rho \log n)$; and (2) the optimum number of pivots to use is $k^* = \Theta(\rho \log n)$ as well.

It is important to make it clear that in many real-world vector spaces the intrinsic dimension is not the same as the *representational* dimension. For example a plane embedded in a 50-dimensional space has intrinsic dimension 2 and representational dimension 50. This is in general the case of real applications, where the data is clustered, and it has lead to attempts to measure the intrinsic dimension such as the concept of "fractal dimension" (Faloutsos and Kamel, 1994). Despite that no proximity search technique can cope with intrinsic dimension higher than 20, much higher representational dimensions can be handled by dimensionality reduction techniques (Faloutsos and Lin, 1995; Cox and Cox, 1994; Hair et al., 1995). Relaxed techniques can be used for "approximate proximity matching", as in (Arya et al., 1994) and (Yianilos, 2000).

3.3. Space Considerations

However, there is an additional factor that we have disregarded up to now. As we use more pivots, our space requirements (i.e. storing kn coordinates) increases. Just storing a few hundreds of coordinates for each element is very expensive, and as we have seen in Figure 5, the optimal k^* is well beyond that limit for all except very low dimensional spaces.

The conclusion is clear: in most cases we have to use just as much memory as we can, since in practice performance will improve monotonically with k. This gives a new light to see all the other algorithms that use a data structure (like trees) over a pivot-based algorithm: the space used by the data structure, aimed at reducing the extra CPU cost,

could perhaps be better used to store more pivots and hence reduce the number of distance evaluations.

4. Fixed Queries Arrays

Under the light of the previous section, we introduce the Fixed Queries Array, or FQA. The FQA is a simple data structure that stores nothing more than the kn coordinates and performs the same number of distance evaluations of the basic technique. However, the FQA permits sublinear (in n) extra CPU time *without* any space overhead. This has not been achieved up to now. Additionally, the FQA permits to trade number of pivots for precision and hence to optimize the amount of memory that can be used. In practice, this reduces the number of distance evaluations to perform at query time.

For a traditional (exact) searching, one can select between an array and a tree to implement essentially the same idea: binary searching. However, proximity searching algorithms work by backtracking in the tree. The essential idea of the FQA is that the same backtracking can be performed in the array without any extra information and with a small time penalty.

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 of height k. If the leaves of the FHFQT 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 (that is, those whose first h values match the path of values leading to the FHFQT node, of depth h). 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

[1] This has close resemblances to suffix trees and suffix arrays, two text retrieval data structures (Baeza-Yates and Ribeiro-Neto, 1999).

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, it just compares sequences of integers. The net result is that the number of distance evaluations is the same, and the extra CPU time is multiplied by an additional $O(\log n)$ factor. As proved in (Baeza-Yates and Navarro, 1998), the FHFQT has $O(n^\alpha)$ extra CPU complexity ($0 < \alpha < 1$), and this converts into $O(n^\alpha \log n)$ for the FQA. The number of distance evaluations can be made $O(\log n)$ by using $\Theta(\log n)$ pivots.

The construction complexity is $O(nk)$ distance evaluations plus the time to sort the array lexicographically. This is $O(kn \log n)$ time.

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 $p_1 \ldots p_k$, 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 prematurely when the remaining sub array is empty.

Nearest neighbor searching can be done in a similar way. The key is to find the distance r^* from q to its nearest neighbor. We start with an estimation $r^* = \infty$ and reduce it each time a closer element to q is discovered. At each point we perform normal range searching with radius r^*. At the end we have in r^* the distance from q to its nearest neighbors and we have already visited all of them. In order to quickly find elements that are close to q, we should start visiting, for each pivot p, the branch labeled $d(q,p)$, then $d(q,p) - 1$, then $d(q,p) + 1$, then $d(q,p) - 2$, then $d(q,p) + 2$, and so on, until $d(q,p) + r^*$, hoping that r^* will be reduced by that time. This is easily extended to find the K nearest neighbors. In this case we keep a priority queue of the current K nearest neighbors sorted by distance. We insert newly found neighbors as we find them, and r^* is the distance from q to the farthest of its current K nearest neighbors. Alternative methods to traverse the tree in order to find promising neighbors as quickly as possible are discussed in (Uhlmann, 1991b) and are applicable here as well.

4.2. An Example

Consider the FHFQT of Figure 6. 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 enters, 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).

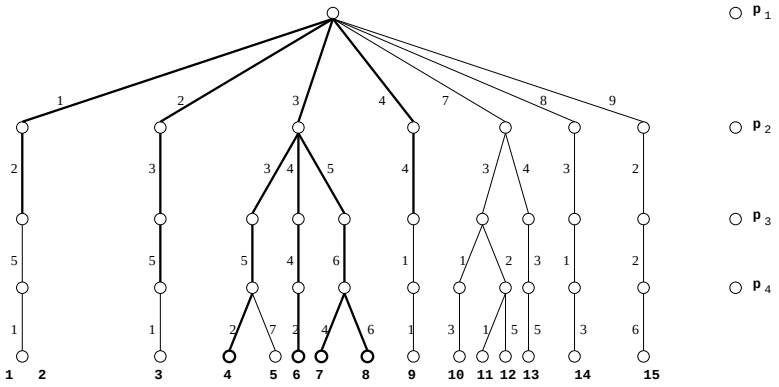


Figure 6. A FHFQT (tree implementation of FQA) for a small example

The equivalent FQA stores the elements in the left-to-right order shown in Figure 6, keeping the four distances for each element. Figure 7 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 $(d(q,p_1),\ldots,d(q,p_4))$, 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 7 illustrates this. With a 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 the 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

Figure 7. 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.

4.3. 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 $b = k \cdot b_s$ 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. The most significant bits are assigned to the first pivots, so 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 intervals to branches in the tree. We envision at least two possible discretization schemes.

Fixed slices: FSFQA For each pivot, independently, we obtain $D_{max} = \max\{d(p_i, u)\}$ and $D_{min} = \min\{d(p_i, u)\}$ for $u \in \mathbb{U} - \{p_1 \ldots p_k\}$. The range $D_{max} - D_{min}$ is divided then in 2^{b_s} parts, and each binary number x is associated to the interval $[D_{min} + x\ (D_{max} - D_{min})/2^{b_s}, D_{min} + (x+1)\ (D_{max} - D_{min})/2^{b_s})$. The idea is that the range of possible values is split in 2^{b_s} slices of the same width, although the number of points lying in each slice may vary (and can be even zero).

Fixed quantiles: FQFQA For each pivot we determine the $b_s - 1$ uniform quantiles that divide the set of distances into b_s equal sized subsets, and assign one quantile to each value. The procedure ensures that in each interval there are exactly $n/2^{b_s}$ points.

5. Experimental Results

In which follows we present experiments on the diverse aspects of the data structure proposed and on how it compares against others. We have selected a sample of 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. We use a database of up to 100,000 elements and 4 to 20 dimensions.

5.1. Use of Memory

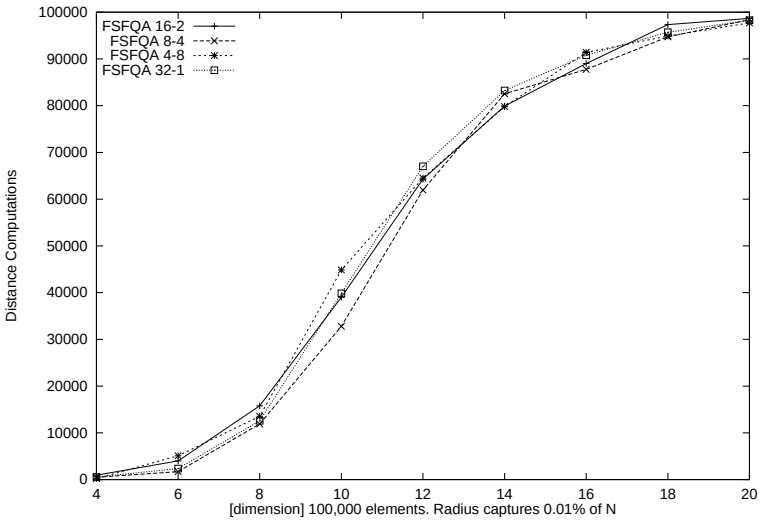
For a fixed amount of memory there are many possible combinations of pivots and resolution for both FSFQA and FQFQA. For example if we have 32 bits for each database point, then we can choose to have 32 1-bit pivots, or 16 2-bit pivots, or 8 4-bit pivots, etc. We try to figure out the best combination and the amount of difference between them.

Another unclear issue is what is the best scheme for FQA: FSFQA or FQFQA. In the graphs the schemes are named "FSFQA/FQFQA $h-b$", where h is the number of pivots used and b is the number of bits per pivot. If the same memory is used then $h \cdot 2^b$ is constant.

In Figure 8 we observe that for a small amount of memory, the difference between schemes is negligible for the FSFQA. However, using 256 bits (8 words), the best is 4 bits per pivot in almost every dimension. Figure 9 shows the same behavior for the FQFQA, the optimal selection being 4 bits per pivot. As can be seen, both schemes give very similar results, so we consider only FQFQA from here on.

Figure 10 compares the FQFQA using 4 bits per pivot, for 32 and 256 bits per element, against other data structures. We have included the FHFQT (height 8, using fixed slices of width 0.1), the MVPT (with arity 16), LAESA (8 pivots), SPA (4 pivots), GNAT (arity 64) and SAT. We gave to FHFQT, LAESA and SPA the same amount of memory as for the FQA of 256 bits (since with 32 bits none of these structures can perform reasonably well), and this determines the height of the FHFQT and the number of pivots for LAESA and SPA. On the other hand, GNATs, SATs and MVPTs use a fixed amount of memory (about 64 bits per element with a very careful implementation).

It is clear that the FQA makes the best use of the available memory, in any of its two variants (which are not very different indeed). With respect to Voronoi type data structures (GNAT and SAT), these are more resistant to high dimensions, but the FQA (and other pivot based algorithms) beats them in any dimension provided enough memory. This shows how important is to make good use of the available memory.



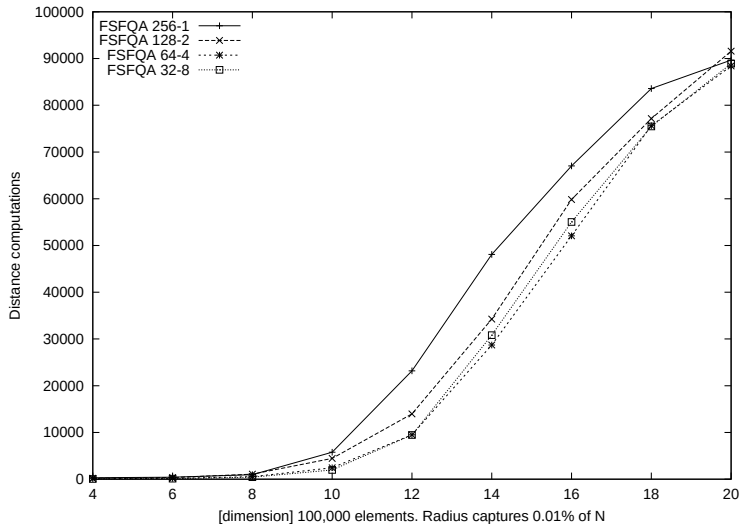


Figure 8. FSFQA using 32 bits (top) and 256 bits (bottom), for several combinations of resolution/pivots and a fixed amount of memory.

5.2. Search Technique

The results presented in the preceding experiments measure only distance computations. We obtain the same results using either the FQA search algorithm or a plain array and a sequential search over the array. If the d distance is expensive to compute, the overall complexity will be driven by the figures obtained using only the number of distance

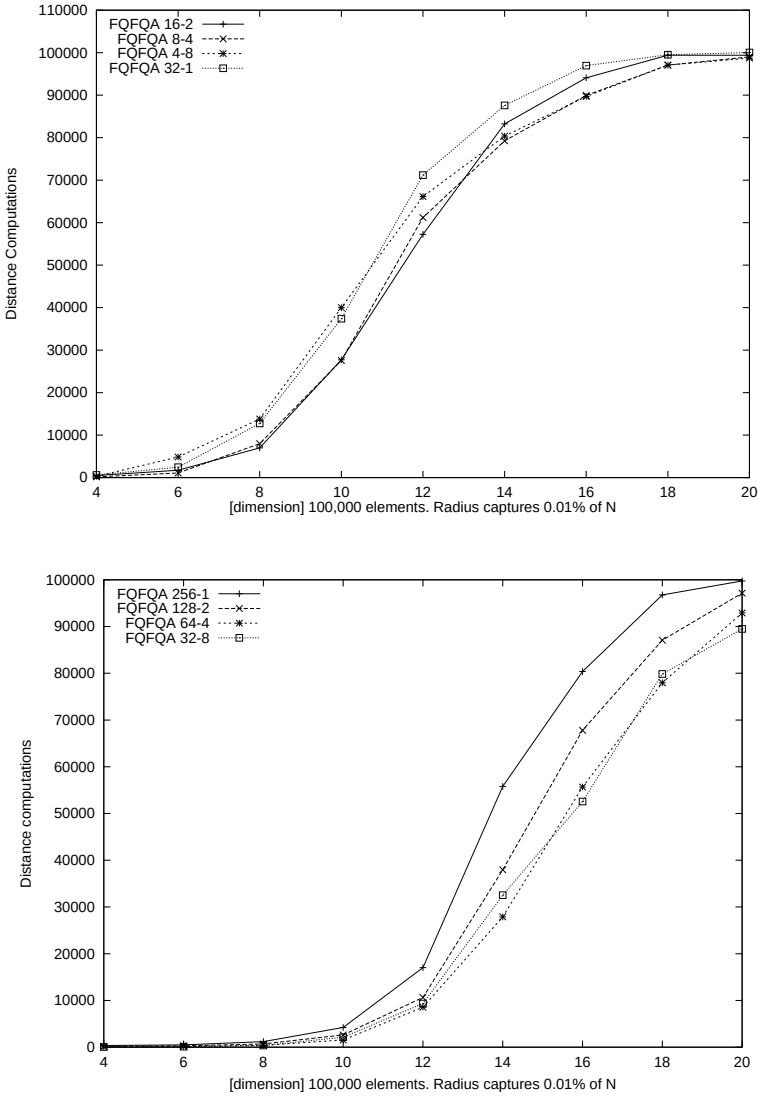


Figure 9. FQFQA using 32 bits (top) and 256 bits (bottom), for several combinations of resolution/pivots and a fixed amount of memory.

computations. In many realistic setups, the side computations cannot be deprecated.

In Figure 11 (top) the total elapsed time for the "sequential" (i.e. a linear pass over the array) and "recursive" (i.e. our backtracking algorithm) versions of the FQA is shown. Note that, as the dimension increases, the differences between both approaches become larger. The same is true if we increase the size of the database, as can be

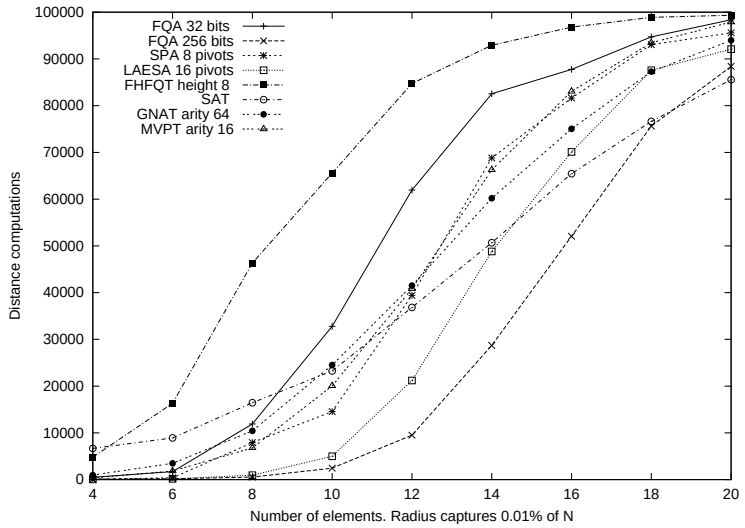


Figure 10. FQA other data structures for different amounts of memory and increasing dimension.

expected when comparing a linear and a sublinear algorithm. Note that the number of distance computations is exactly the same for both implementations.

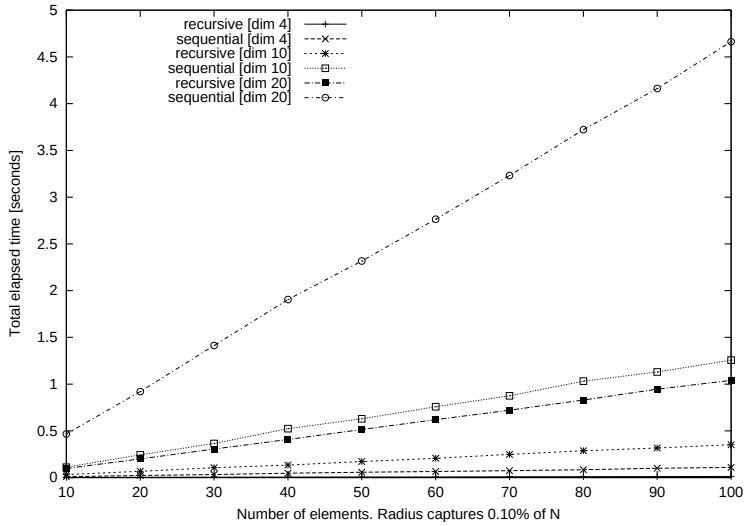
We have also included a least squares estimation of the running time under the model $t = cn^{\alpha}$, showing that the recursive algorithm is sublinear on n, despite that α tends to 1 as the dimension grows.

5.3. A Real-World Example

In this section we aim at studying the performance of the FQA in a real application. We also use the test to determine which is the gain of the FQA with respect to other data structures in total CPU time (not only counting distance evaluations). This test has little sense in the synthetic experiments because the L_2 distance in up to 20 dimensions is much easier to evaluate than most real world distances.

In several applications of computer vision and image processing the user wants to locate a subimage inside a larger image. The reference image may be a frame of a sequence, or a static image. The subimages are sized 16×16 to 32×32 pixels. The representational dimension of the vectors is 256 and 1024 respectively. The distance used to match subimages is the Euclidean distance or the L_∞ distance.

We ran an experiment (see Figure 12) for finding 15×15 subimages inside a 256×256 pixels image. This is equivalent to searching for a 225-dimensional vector in a 60,000 elements database. The search



Dimension	Sequential	Recursive
4	$.0003n^{1.13}$	$.0018n^{0.26}$
10	$.0012n^{1.04}$	$.0020n^{0.87}$
20	$.0120n^{0.99}$	$.0086n^{0.98}$

Figure 11. Total elapsed time for varying database size. The plot shows the behavior of the recursive and sequential implementations of the FQFQA 64-8 in several dimensions.

retrieved about 6 database elements (0.01 % of the database) per query. The results were averaged over 300 random queries. The machine is a Pentium III 450 Mhz with two processors and 500 Mb of RAM, running Linux Kernel 2.2.12.

For this example the FQFQA is faster than all the other data structures, even taking less space (we had better results with 8 bits per pivot this time). It can easily trade memory for time. When moving from 16 to 32 pivots (from 1 Mb to 2 Mb) the speedup is about 25%, going from 32 to 64 pivots (from 2 to 4 Mb) gives a smaller speedup, of about 3%. We used 8 pivots for SPA, 16 pivots for LAESA and height 8 for FHFQT. The memory is measured as the amount required by the process at runtime.

Index	Construction (distances)	Query time (msecs)	Query time (distances)	Memory
FQA 64-8	3.5 M	14.167	245	4.0 Mb
FQA 32-8	2.1 M	14.556	285	2.0 Mb
FQA 16-8	1.1 M	18.333	414	1.0 Mb
SPA	0.5 M	22.833	548	4.0 Mb
FHFQT	0.5 M	26.111	594	8.1 Mb
MVPT	1.9 M	35.444	915	5.0 Mb
GNAT	7.1 M	62.722	1,702	1.2 Mb
SAT	5.5 M	175.222	4,554	3.6 Mb
LAESA	1.1 M	196.611	335	4.0 Mb

Figure 12. Finding 15×15 subimages inside a 256×256 pixels image. The database has about 60,000 vectors of dimension 225. Construction complexity is measured in (millions) of distance computations. Query complexity is measured in milliseconds as well as in distance computations.

6. Conclusions and Future Work

We have presented a new data structure, the Fixed Queries Array (FQA), for proximity searching in metric spaces. The FQA belongs to the family of pivot based algorithms, the most popular one. We have argued that the most important parameter governing the performance of those algorithms is the number k of pivots used. Therefore, the amount of available memory is a crucial parameter because we need to store kn coordinates.

The essential advantage of the FQA is that it makes good use of the available memory. First, it is the only structure that permits a sublinear amount of extra CPU time *without* using any extra information, so the space that other data structures use for this purpose can be used to accommodate more pivots. Second, it permits to trade number of pivots for their precision, in order to make optimal use of the available memory.

The FQA is experimentally shown to be a simple yet effective structure for this problem, as it compares well with other state of the art approaches. Its simplicity makes it very appealing for practitioners seeking for a black box data structure to plug in their applications.

We are currently working on dynamic and secondary memory aspects of this data structure. For example, inserting an element is costly in a sorted array, but a slightly more complex scheme with linked

blocks solves the problem at negligible extra space overhead. Secondary memory problems refer to the design of efficient access paths for this data structure when it is stored on disk.

The code for the FQAs is available upon request, at `elchavez@zeus.ccu.umich.mx`

References

Arya, S., D. Mount, N. Netanyahu, R. Silverman, and A. Wu: 1994, 'An optimal algorithm for approximate nearest neighbor searching in fixed dimension'. In: *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms (SODA'94)*. pp. 573–583.

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

Baeza-Yates, R.: 1997, 'Searching: an algorithmic tour'. In: A. Kent and J. Williams (eds.): *Encyclopedia of Computer Science and Technology*, Vol. 37. Marcel Dekker Inc., pp. 331–359.

Baeza-Yates, R., W. Cunto, U. Manber, and S. Wu: 1994, 'Proximity Matching Using Fixed-Queries Trees'. In: *Proc. 5th Combinatorial Pattern Matching (CPM'94)*. pp. 198–212.

Baeza-Yates, R. and G. Navarro: 1998, 'Fast Approximate String Matching in a Dictionary'. In: *Proc. 5th Symposium on String Processing and Information Retrieval (SPIRE'98)*. pp. 14–22.

Baeza-Yates, R. and B. Ribeiro-Neto: 1999, *Modern Information Retrieval*. Addison-Wesley.

Bentley, J.: 1975, 'Multidimensional binary search trees used for associative searching'. *Comm. of the ACM* **18**(9), 509–517.

Bozkaya, T. and M. Ozsoyoglu: 1997, 'Distance-based indexing for high-dimensional metric spaces'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 357–368. Sigmod Record 26(2).

Brin, S.: 1995, 'Near neighbor search in large metric spaces'. In: *Proc. 21st Conference on Very Large Databases (VLDB'95)*. pp. 574–584.

Chávez, E., J. Marroquín, and R. Baeza-Yates: 1999a, 'Spaghettis: an Array Based Algorithm for Similarity Queries in Metric Spaces'. In: *Proc. 6th Symposium on String Processing and Information Retrieval (SPIRE'99)*. pp. 38–46.

Chávez, E., J. Marroquín, and G. Navarro: 1999b, 'Overcoming the Curse of Dimensionality'. In: *European Workshop on Content-Based Multimedia Indexing (CBMI'99)*. pp. 57–64. `ftp://garota.fismat.umich.mx/pub/users/-elchavez/fqa.ps.gz`.

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

Ciaccia, P., M. Patella, and P. Zezula: 1997, 'M-tree: an efficient Access Method for Similarity Search in Metric Spaces'. In: *Proc. of the 23rd Conference on Very Large Databases (VLDB'97)*. pp. 426–435.

Clarkson, K.: 1999, 'Nearest neighbor queries in metric spaces'. *Discrete Computational Geometry* **22**(1), 63–93.

Cox, T. and M. Cox: 1994, *Multidimensional Scaling*. Chapman and Hall.

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

Faloutsos, C. and I. Kamel: 1994, 'Beyond uniformity and independence: analysis of R-trees using the concept of fractal dimension'. In: *Proc. 13th ACM Symposium on Principles of Database Principles (PODS'94)*. pp. 4–13.

Faloutsos, C. and K. Lin: 1995, 'Fastmap: a fast algorithm for indexing, data mining and visualization of traditional and multimedia datasets'. *ACM SIGMOD Record* **24**(2), 163–174.

Guttman, A.: 1984, 'R-trees: a dynamic index structure for spatial searching'. In: *Proc. ACM SIGMOD International Conference on Management of Data*. pp. 47–57.

Hair, J., R. Anderson, R. Tatham, and W. Black: 1995, *Multivariate Data Analysis with Readings*. Prentice-Hall, 4th edition.

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

Micó, L., J. Oncina, and E. Vidal: 1994, '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.

Navarro, G.: 1999, 'Searching in metric spaces by spatial approximation'. In: *Proc. 6th Symposium on String Processing and Information Retrieval (SPIRE'99)*. pp. 141–148.

Nene, S. and S. Nayar: 1997, 'A simple algorithm for nearest neighbor search in high dimensions'. *IEEE Trans. on Pattern Analysis and Machine Intelligence* **19**(9), 989–1003.

Shapiro, M.: 1977, 'The choice of reference points in best-match file searching'. *Comm. of the ACM* **20**(5), 339–343.

Uhlmann, J.: 1991a, 'Implementing metric trees to satisfy general proximity/similarity queries'. Manuscript.

Uhlmann, J.: 1991b, 'Satisfying general proximity/similarity queries with metric trees'. *Information Processing Letters* **40**, 175–179.

Vidal, E.: 1986, 'An algorithm for finding nearest neighbors in (approximately) constant average time'. *Pattern Recognition Letters* **4**, 145–157.

Yianilos, P.: 1993, 'Data structures and algorithms for nearest neighbor search in general metric spaces'. In: *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms (SODA'93)*. pp. 311–321.

Yianilos, P.: 1999, 'Excluded Middle Vantage Point Forests for Nearest Neighbor Search'. In: *DIMACS Implementation Challenge, ALENEX'99*. Baltimore, MD.

Yianilos, P.: 2000, 'Locally Lifting the Curse of Dimensionality for Nearest Neighbor Search'. In: *Proc. 11th ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*. pp. 361–370.