NB-Tree: An Indexing Structure for Content-Based Retrieval in Large Databases

Manuel J. Fonseca*, Joaquim A. Jorge
Department of Information Systems and Computer Science
INESC-ID/IST/Technical University of Lisbon
R. Alves Redol, 9, 1000-029 Lisboa, Portugal
mjf@inesc-id.pt, jorgej@acm.org

Abstract

Many indexing approaches for high–dimensional data points have evolved into very complex and hard to code algorithms. Sometimes this complexity is not matched by increase in performance. Motivated by these ideas, we take a step back and look at simpler approaches to indexing multimedia data. In this paper we propose a simple, (not simplistic) yet efficient indexing structure for high–dimensional data points of variable dimension, using dimension reduction. Our approach maps multidimensional points to a 1D line by computing their Euclidean Norm. In a second step we sort these using a B^+-Tree on which we perform all subsequent operations. We exploit B^+-Tree efficient indexed sequential search to develop simple, yet performant methods to implement point, range and nearest-neighbor queries.

To evaluate our technique we conducted a set of experiments, using both synthetic and real data. We analyze creation, insertion and query times as a function of data set size and dimension. Results so far show that our simple scheme outperforms current approaches, such as the Pyramid Technique, the A-Tree and the SR-Tree, for many data distributions. Moreover, our approach seems to scale better both with growing dimensionality and data set size, while exhibiting low insertion and search times.

1 Introduction

In recent years, increasing numbers of computer applications in CAD, geography, biology, medical imaging, etc., access data stored on large databases. A common feature to such databases is that

*Corresponding author.

objects are described by vectors of numeric values, known as *feature vectors*, which map individual instances to points in a high dimensional vector space.

An important functionality that should be present in such applications is similarity search, *i.e.* finding a set of objects similar to a given query. The similarity between complex objects is not measured on their contents directly, since this tends to be expensive. Rather we use their feature vectors, assuming that features are "well-behaved", that is, similar objects have feature vectors that are near in hyperspace and vice-versa. This way, searching objects by similarity in a database becomes a nearest neighbor search in a high-dimensional vector space, followed by similarity tests applied to the ten resulting points.

To support processing large amounts of high–dimensional data, a variety of indexing approaches have been proposed in the past few years. Some of them are structures for low–dimensional data that were adapted to high–dimensional data spaces. However, such methods while providing good results on low–dimensional data, do not scale up well to high–dimensional spaces. Recent studies [17] show that many indexing techniques become less efficient than sequential search, for dimensions higher than ten. Other indexing mechanisms are incremental evolutions from existing approaches, where sometimes, the increased complexity does not yield comparable enhancements in performance. Other indexing techniques based on dimension reduction return only approximate results. Finally, structures combining several of the previous approaches have emerged. Often, the corresponding algorithms are very complex and unwieldy.

Our interest in this problem derives from the development of effective and efficient systems for content-based retrieval of technical drawings. In the approach we presented in [8], technical drawings are described using topology graphs that include shape and spatial information. On a second step, graphs are converted in feature vectors by computing and combining eigenvalues from the adjacency matrix of the graph. The dimensionality of the resulting feature vectors will depend of the complexity of the technical drawing. More complex drawings will produce descriptors of higher dimension, while simple drawings will produce lower dimension feature vectors.

All indexing structures studied so far, only support data sets of fixed dimension. However, in some application domains, as the one described before, the dimension of feature vectors can vary from object to object and the maximum dimension can not be predicted in advance. In such scenarios, current indexing structures will define a (maximum) fixed dimension for the data space and feature vectors of smaller dimensions will be padded with zeros. However, if we are to insert new feature vectors of larger than maximum dimension, the indexing structure must be rebuilt to

accommodate the new data.

The increasingly complex data structures and specialized approaches to high-dimensional indexing make it difficult to ascertain wether there might be a reasonably fast and general approach to address variable dimension data. We believe there might be some merit in taking a step back and looking at simpler approaches to indexing such data.

Unlike other existing approaches, we use a very simple dimension reduction function to map high–dimensional points into a one–dimensional value. Our approach is motivated by three observations. First, real data sets tend to have a lot of clusters distributed along the data space. Thus, the Euclidean norms of points tend to be "evenly" distributed (see section 4). Second, the set of resulting nearest neighbors will lie inside a narrow range of norm values, thus reducing the number of data points to examine during search. Third, some application domains need to manipulate large amounts of variable dimension data points, which calls for an approach that can handle variable data in a natural manner.

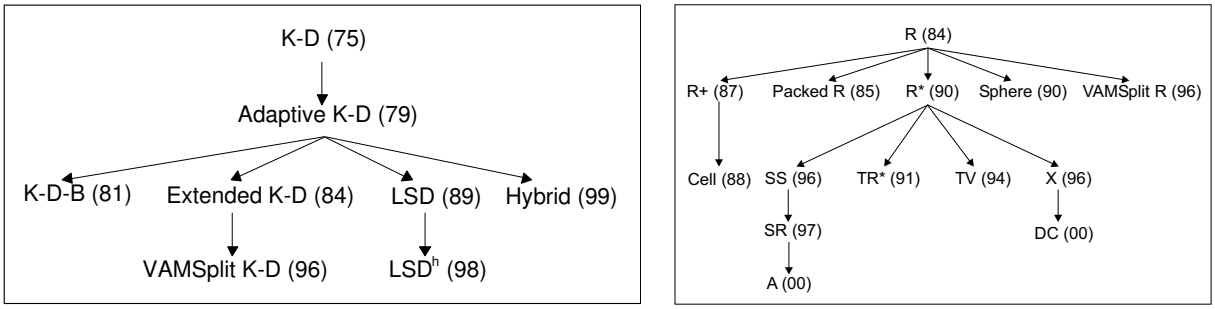
Based on these requirements, we developed the NB-Tree[1], an indexing technique based on a simple, yet efficient algorithm to search points in high–dimensional spaces with variable dimension, using dimension reduction. Multidimensional points are mapped to a 1D line by computing their Euclidean Norm. In a second step we sort these mapped points using a B^+-Tree on which we perform all subsequent operations. Thus, the NB-Tree can be implemented on existing DBMSs without additional complexity.

Our approach supports all typical kinds of queries, such as point, range and nearest neighbor queries (KNN). Also, our method provides fast and accurate results, in contrast to other indexing techniques, which return approximate results. Moreover, for some methods [20] when accuracy increases, their performance decreases.

We implemented the NB-Tree and evaluated its performance against more complex approaches, such as, the Pyramid Technique, the A-Tree and the SR-Tree. We conducted a set of experiments, using synthetic and real data, to analyze creation, insertion and query times as a function of data set size and dimension. Results so far show better results for our method, for many data distributions. Moreover, our approach seems to scale better both with growing dimensionality and data set size, while exhibiting low insertion and search times.

The rest of the paper is organized as follows. In the next section we give an overview of the related work in high–dimensional indexing structures. Section 3 explains the basic idea of the NB-

[1] **N**orm + $\mathbf{B}^+$-Tree = **NB**-Tree



a) K-D-Tree b) R-Tree

Figure 1: Family trees of most common indexing structures.

Tree, the algorithms for insertion and search and the complexity analysis. In section 4 we present experimental studies to characterize the different data sets used for evaluation. Section 5 describes our experimental evaluation and shows performance results comparing our structure to the best recently published approaches, such as the SR-Tree, the A-Tree and the Pyramid Technique. We conclude the paper by discussing the results and presenting further work directions in our algorithm and related areas.

2 Related Work

The indexing techniques developed so far can be classified into two categories. One that includes all structures derived from the K-D-Tree (see Figure 1.a) and the other composed by derivatives of the R-Tree (see Figure 1.b). The main difference between these two categories lies in the approach to dividing the data space. Structures in the first category use space–partitioning methods that divide the data space along predefined hyper-planes regardless of data distribution. The resulting regions are mutually disjoint, with their union being the complete space. Structures from the second class use data–partitioning methods, which divide the data space according to their distribution. This can yield possible overlapping regions. Besides these two categories, there are other techniques combining several methodologies to improve performance.

White and Jain presented the VAM-Split K-D-Tree [18], which is an extension to the K-D-Tree [14] to improve the efficiency and to reduce the storage space. The main difference between these two trees is in the way they split the data space. While the K-D-Tree uses the 50% quantile, the VAM-Split K-D-Tree uses the maximum variance and the median. This tree outperforms the K-D-Tree and the SS-Tree, but while these two are dynamic index structures, the VAM-Split K-D-Tree is a static structure, *e.g.* all data items must be available at creation time.

In [9] Henrich proposed the LSD^h-Tree as an improvement to the LSD-Tree [10]. The paging routine of the LSD^h-Tree keeps part of the tree in main memory and stores some sub-trees in disk when the structure becomes to large, keeping the tree balanced. The LSD^h-Tree combines the low fanout from K-D-Trees with the advantage that R-Trees have in not covering empty space. This structure is slightly better than the X-Tree.

The Hybrid-Tree [5] was introduced by Chakrabarti in 1999 to combine the advantages from space–partitioning structures and data–partitioning structures. The Hybrid-Tree always splits a node using a single dimension to guarantee that the fanout is independent of data dimension. However, this method allows overlapping regions as data–partitioning structures do. This structure outperforms both the SR-Tree and the hB-Tree [13].

White and Jain presented the SS-Tree [19], an R-Tree-like index structure that uses minimum bounding spheres (MBSs) instead of minimum bounding rectangles (MBRs) in the directory. Even though the use of spheres reduce the overlapping of regions and consequently the SS-Tree outperforms the R*-Tree, spheres tend to overlap for high–dimensional spaces.

Recently, Katayama proposed the SR-Tree [11], an improvement to the SS-Tree combining concepts from both the R*-Tree and SS-Tree approaches. This structure uses both MBRs and MBSs as an approximation in the directory. This way, the region spanned by a node is the intersection of the MBR and MBS associated to that node, thus reducing region overlap between sibling nodes. This structure has been shown to outperform both the R*-Tree and the SS-Tree. In [15] Sakurai presents an experimental evaluation, using non-uniform data, where the SR-Tree outperforms the VA-File [17].

In [18] White introduced the VAMSplit R-Tree, which is a static index structure that splits the data space depending on the maximum data variance, like the VAM-Split K-D-Tree. While this structure seems to outperform the R-tree, it has the shortcoming that all data need to be known *a priori*. Thus it is not suited for dynamically varying data sets and environments

Berchtold, Keim and Kriegel proposed the X-Tree [4], an index structure adapting the algorithms of R*-Tree to high–dimensional data. The X-Tree uses two techniques to deal with high-dimensional data: First, it introduces an overlap–free split algorithm, which is based on the split history of the tree. Second, where the overlap–free split algorithm would lead to an unbalanced directory, the X-Tree omits the split and the according directory node is enlarged becoming a super-node. The X-Tree outperforms both the R*-Tree and the TV-Tree [12].

Recently, Sakurai proposed the A-Tree [15], which is an index structure for high-dimensional

data that introduced the notion of relative approximation. The basic idea of the A-Tree is the use of Virtual Bounding Rectangles to approximate minimum bounding rectangles or objects. Thus, on one node of the tree we have information about the exact position of the region MBR and an approximation of the relative position of its sons. The authors claim that the A-Tree outperforms the VA-File and the SR-Tree.

In 1998 Weber et al. [17] proposed the VA-File, a method to search points of high dimension. The authors show that existing structures (R*-Tree and X-Tree) are outperformed on average by a simple sequential scan if the dimension exceeds around ten. So, instead of developing another indexing structure they proposed to speed-up the sequential scan. The basic idea of the VA-File is to keep two files; one with an approximate version of data points and other with the exact representation. When searching points, the approximation file is sequentially scanned with some look-ups to the exact file whenever it is necessary. The VA-File outperforms both the R*-Tree and the X-Tree when the dimension is higher than six, but its performance is very sensitive to data distribution.

Berchtold et al.' Pyramid Technique [2] uses a special partitioning strategy optimized for high–dimensional data. Its basic idea is to perform a dimension reduction allowing the use of efficient uni-dimensional index structures to store and search data. The Pyramid Technique divides the data space into 2D pyramids whose apexes lie at the center point. In a second step, each pyramid is cut into several slices parallel to the basis of the pyramid forming the data pages. The Pyramid Technique associates to each high–dimensional point a single value, which is the distance from the point to the top of the pyramid, according to a specific dimension (j_{max}). This value is later used as a key in the B^+-Tree. Points in the same slice will have the same identification value. As a result, for skewed distributions, many points in the B^+-Tree will be indexed by the same key. The Pyramid Technique outperforms the X-Tree and the sequential scan using uniformly distributed data. Although the Pyramid Technique was developed mainly for uniformly distributed data, authors suggested an extended version to handle real (skewed) data. This version works by shifting the center point to the barycenter of the data. However, in a dynamic scenario, this implies the recalculation of all index values, *i.e.* redistribution of points among the pyramids and reconstruction of the B^+-Tree, each time the center of the cluster changes. We have receded from Berchtold's work by keeping the dimension reduction approach and focusing on reducing data-structure traversal and book keeping overheads. These steps result in simpler algorithms and data structures, independent of data distribution, which are also more flexible in accommodating dynamically changing data of

varying dimension.

In [3] Berchtold et al. described a new contribution to the near neighbor search through precomputation and indexing the solution space. First, they precompute the result of any nearest neighbor search, which corresponds to computing the Voronoi cell of each data point. In a second step, they store approximations of each Voronoi cells in an indexing structure, the X-Tree. Subsequently, searching for the nearest neighbor maps to a simple point query on the index structure. Although the complexity of these approximations increases quickly with the data dimension, this new technique outperforms the X-Tree.

More recently, Berchtold et al. proposed the IQ-Tree [1] which combines a compression method with an index structure trying to join the best of both worlds. The IQ-Tree is a three-level structure, with a regular directory level, a compression level and a third level that contains the actual data. While the VA-File uses a fixed number of bits to compress all partitions, the IQ-Tree uses a different compression scheme for each partition depending on the density of data points. This structure outperforms both the X-Tree and the VA-File.

In [16] Shepherd et al. presented an overview of an efficient file–access method for similarity searching in high–dimensional spaces, named CurveIx. The basic idea is to order the d–dimensional space in many ways, with a set of (one–dimensional) space–filling curves. Each curve constitutes a mapping from $R^d \rightarrow R^1$, yielding a linear ordering of all points in the data set. During retrieval only points whose location is close to the curve-location of the query are considered. Their approach uses standard one–dimensional indexing schemes such as the B-Tree to perform searching. Close points along the space–filling curve tend to correspond to close points in the d–dimensional feature space. However, some near neighbors may be mapped far apart along a single space–filling curve. As a result, this method does not have an accuracy of 100%, *i.e.* some near neighbors may be ignored.

More recently, Yu et al. proposed a new index scheme, called iMinMax(θ) [20], that maps high–dimensional points to single dimension values determined by their maximum or minimum coordinate values. By varying the θ value they can optimize the iMinMax structure to different distributions of data sets. As other dimension reduction methods, this scheme also uses the B^+-Tree to index the resulting single dimension points. The iMinMax structure was mainly developed to address window search (range queries) while still supporting approximate KNN search (accuracy less than 100%) at the expense of increasing runtimes for higher accuracy. This approach outperforms both the VA-File and the Pyramid Technique for range queries.

The research work outlined above highlights two main patterns in previous developments. First, there is no total order connecting any two algorithms so far developed. This suggests both the need to do some research work on developing reference data sets representative of significant problem domains to ensure commensurable results. Second, due to the diversity of problem domains, it is difficult to establish emerging techniques as related to previous work. Sometimes, even when reference implementations of some algorithms exist, problems with existing code prevent other researchers from duplicating experimental results. It becomes difficult to compare results either when the reference implementation does not provide correct results or the programs terminate abnormally on large data sets.

In the next section we describe our work on a simple data structure, the NB-Tree, that attempt to overcome some of the limitations we highlighted.

3 The NB-Tree Structure

The NB-Tree provides a simple and compact means to indexing high–dimensional data points of variable dimension, using a light mapping function that is computationally inexpensive. The basic idea of the NB-Tree is to use the Euclidean norm value as the index key for high–dimensional points. Thus, values resulting from the dimension reduction can be ordered and later searched in the resulting one–dimensional structure. To index data points sorted by their Euclidean norm we use the B^+-Tree, since it is the most efficient indexing structure for one dimension and also, because it is supported by all commercial DBMSs.

The use of the Euclidean norm as a mapping function from $R^D \rightarrow R^1$, assures that near high–dimensional points will have near Euclidean norms. Consequently, when performing a query (of any type) the system only examine points whose norm is in the neighborhood of the query point norm. Moreover, the B^+-Tree has the particularity that its leaves are linked as an ordered list. Thus, walking sequentially through all the elements of the B^+-Tree is a costless operation.

3.1 Creating an NB-Tree

In the following discussion, we consider that the data space is normalized to the unit hypercube $[0..1]^D$ and that an arbitrary data point in the space is defined as $P = (p_0, p_1, \cdots, p_{D-1})$.

To create an NB-Tree we start by computing the Euclidean norm of each D–dimensional point from the data set, using the Formula:

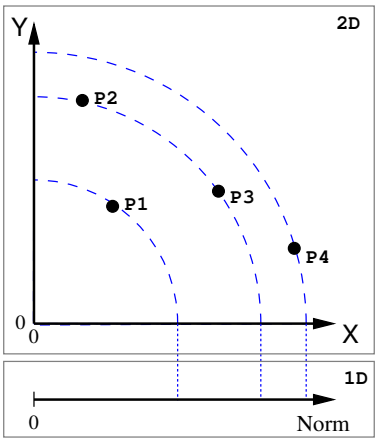


Figure 2: Dimension Reduction.

$$\|P\| = \sqrt{p_0^2 + p_1^2 + \cdots + p_{D-1}^2}$$

D–dimensional points are then inserted into a B$^+$-Tree, using the norm as key. After inserting all points we get a set of D–dimensional data ordered by norm value. Figure 2 shows an example of dimension reduction for 2D data points. As we can see, with this mapping function, different points can have the same key value. On the other hand near data points have similar keys (norms).

3.2 Searching

The searching process in the NB-Tree starts by computing the norm of the query point. Then we perform a search in the 1–dimensional B$^+$-Tree. The search steps will depend of the query type. Current indexing structures usually support three types of queries. The first is **Point Query**, which checks if a specific point belongs or not to the database. The second type, **Range Query**, returns the points inside a specific range of values. In our case that range will be specified by an hyper-ball. Finally, the ***k*-NN Query** (k Nearest Neighbors) returns the k nearest neighbors of the query point. This is the most often-used query in content–based retrieval.

The next subsections describe how each type of query is implemented by the NB-Tree.

3.2.1 Point Query

After computing the Euclidean norm of the query point, we use this value as key to search the B$^+$-Tree. For the point(s) returned by the B$^+$-Tree (if any) we compute the distance to the query point. If we get a distance of zero, it means the query point exists in the NB-Tree.

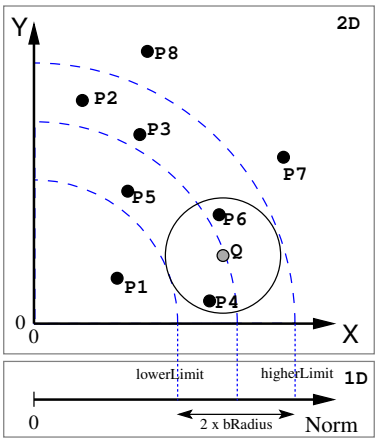


Figure 3: 2D Range query example.

3.2.2 Range Query

The first step for a range query, after computing the query norm, is to compute the lower and the higher bounds on the norm for the corresponding query.

$$lowerLimit = \|Q\| - bRadius$$

$$higherLimit = \|Q\| + bRadius$$

where $bRadius$ is the radius of the hyper–ball, which spans the search range.

After that, we search the B$^+$-Tree for the point with $lowerLimit$ as key. After finding the first point we sequentially scan the B$^+$-Tree until we reach the $higherLimit$. As we said before scanning the B$^+$-Tree sequentially is an inexpensive operation. For each point with a norm value in the interval $[lowerLimit, higherLimit]$, we compute the distance to the query point. If this distance is smaller than or equal to the $bRadius$, the point is added to the list of points to be returned. This list is ordered by distance to the query point. Figure 3 illustrates a ball query in 2D, where we can see that an interval in the 1D space corresponds to part of a D–dimensional hypersphere. Figure 4 lists the pseudo-code to implement a ball query.

3.2.3 k-NN Query

Before we start describing our k-NN algorithm we have to define some relevant concepts: **higherLimit** and **lowerLimit** are the norm values used to define the upper and lower bounds to search for nearest neighbors; **delta** - value by which the previous limits are incremented at each step; **priority list** - list with k nodes, where we store k points that temporarily store for intermediate results. The last element of this list is the farthest nearest neighbor so far. Thus, when we want

```
list* ballQuery(query, bRadius)
{
    qNorm = computeEuclideanNorm(query);
    lowerLimit = qNorm - bRadius;
    higherLimit = qNorm + bRadius;
    point = btree->search(lowerLimit);
    // after finding the first point the
    // search is a sequential scanning
    while (pointNorm <= higherLimit) {
        dist = dist2Query(point, query);
        if (dist <= bRadius)
            pointList->addPoint(point);
        point = btree->nextPoint();
    }
    return pointList;
}
```

Figure 4: Range Query pseudo-code.

to insert a new neighbor into the list, we just have to compare its distance against the last.

We start the nearest neighbor search by locating a point in the B^+-Tree with a key equal (or near) the norm of the query point. After this positioning in the 1D line, we execute a set of iterative steps until we get all the desired results. Since the B^+-Tree has its leaves linked sequentially, we only navigate at the leaves level, as if we were using a double linked ordered list. Below we describe all the steps to achieve the final k-NN points.

After positioning in the 1D line we define the *upperLimit* and the *lowerLimit* based on the *delta* value. Next we go through all points until we reach the *upperLimit* or the point whose norm follows the limit. In this case the *lowerLimit* is changed to keep the symmetry to the query. After, we do the same for the *lowerLimit*. During this forward and backward advance we compute the distance from each point to the query point. As described before, if the distance to the query point is smaller than the distance of the farthest current near neighbor, we store that point in the list. After going up and down, we check whether there are enough points inside the hypersphere of diameter equal to the difference between limits. This iterative process happens until all k neighbors from the list are contained in the previously defined hypersphere.

In summary, we compute the k-NN through an iterative process, where the size of the searching ball increases gradually until we get all the points specified by the query. Figure 5 presents the pseudo-code for the k-NN algorithm.

```
list* knnQuery(query, knn)
{
  qNorm = computeEuclideanNorm(query);
  lowerLimit = qNorm;
  higherLimit = qNorm;
  //grow the ball by going up and down
  // on the B+-Tree
  do {
    point = btree->search(higherLimit);
    higherLimit += delta;
    while (pointNorm <= higherLimit) {
      dist = dist2Query(point, query);
      if (dist < furthest) {
        pointList->deleteLastPoint();
        pointList->addPoint(point);
      }
      point = btree->nextPoint();
    }
    point = btree->search(lowerLimit);
    lowerLimit -= delta;
    while (pointNorm >= lowerLimit) {
      dist = dist2Query(point, query);
      if (dist < furthest) {
        pointList->deleteLastPoint();
        pointList->addPoint(point);
      }
      point = btree->prevPoint();
    }
  } while(!enoughPoints(pointList, knn));
  return pointList;
}
```

Figure 5: k-NN Query pseudo-code.

3.3 Computational Complexity

In this subsection, we present the computational complexity of the NB-Tree insertion, point, range and k-NN query algorithms. We describe the running time of the algorithms as a function of data point dimension and data set size.

It is important for our complexity study to recall that the B^+-Tree has a computational complexity of $O(lgN)$ for insertion and searching algorithms and requires linear space for storage.

Insertion From the description in section 3.1 we conclude that inserting points into a NB-Tree requires two operations: the computation of the Euclidean Norm and an insertion into the B^+-Tree. Table 1 presents the running time for each operation as a function of data set size (N) and data

Operation	N	D
Computation of the Norm	$O(1)$	$O(D)$
Insertion into the B$^+$-Tree	$O(lgN)$	$O(1)$
Total Running Time	$O(lgN)$	$O(D)$

Table 1: Analysis of the NB-Tree Insertion algorithm.

point dimension (D).

Point Query As described in section 3.2.1 the main operations of a point query are the computation of the Euclidean Norm, a search in the B$^+$-Tree and the calculation of a distance. Table 2 shows running times for the point query algorithm.

Operation	N	D
Computation of the Norm	$O(1)$	$O(D)$
Search in the B$^+$-Tree	$O(lgN)$	$O(1)$
Calculation of a distance	$O(1)$	$O(D)$
Total Running Time	$O(lgN)$	$O(D)$

Table 2: Analysis of the NB-Tree Point Query algorithm.

Range Query In Figure 4 the main operations of this algorithm are the computation of the Euclidean Norm, a search in the B$^+$-Tree, the calculation of N distances (in the worst case), the insertion of M points into a B$^+$-Tree (the ones inside the ball, $M << N$) and N sequential steps in

Operation	N	D
Computation of the Norm	$O(1)$	$O(D)$
Search in the B$^+$-Tree	$O(lgN)$	$O(1)$
N x Calculation of a distance	$O(1)$	$O(D)$
M x insertion into a B$^+$-Tree	$O(lgM)$	$O(1)$
N x sequential step	$O(N)$	$O(1)$
Total Running Time	$O(N)$	$O(D)$

Table 3: Analysis of the NB-Tree Range Query algorithm.

the B^+-Tree (in the worst case). The total running time for the range query algorithm is presented in Table 3

k-NN Query From the pseudo-code presented in Figure 5 we can extract the following operations: computation of the Euclidean Norm, a search in the B^+-Tree, the calculation of N distances (in the worst case), the insertion of M points into a list ($M << N$) and N sequential steps in the B^+-Tree (in the worst case). Table 4 shows the running times for the k-NN algorithm.

Operation	N	D
Computation of the Norm	$O(1)$	$O(D)$
Search in the B^+-Tree	$O(lgN)$	$O(1)$
N x Calculation of a distance	$O(1)$	$O(D)$
M x insertion into a list	$O(lgM)$	$O(1)$
N x sequential step	$O(N)$	$O(1)$
Total Running Time	$O(N)$	$O(D)$

Table 4: Analysis of the NB-Tree k-NN Query algorithm.

In Table 5 we present a summary of the complexity analysis. As we can see, the NB-Tree presents a logarithmic running time for insertion and point query and a linear running time for range query and k-NN query, when we consider the size of the data set. If we now consider the dimension, the NB-Tree has a worst-case running time linear with the dimension.

NB-Tree Algorithm	N	D
Insertion	$O(lgN)$	$O(D)$
Point Query	$O(lgN)$	$O(D)$
Range Query	$O(N)$	$O(D)$
k-NN Query	$O(N)$	$O(D)$

Table 5: Summary of the NB-Tree complexity analysis.

4 Data Set Characterization

Before describing our experimental evaluation, we present an experimental study characterizing the different data distributions we used to evaluate our NB-Tree. The first data sets analyzed were of

uniformly distributed data, with independent attributes. After, we analyzed a set of real data and another synthetic data set simulating a data space of variable dimension. We assume that the data space, for all kinds of data sets, is normalized to the unit hypercube $[0..1]^D$.

While data points generated from a uniformly distributed set of coordinates arguably represent any "real" distribution, most published approaches provide performance measurements and comparisons based on such data. Thus it becomes logical to evaluate our indexing structure using these data sets. Moreover, as we will show in the present section, many algorithms (including ours) seem to perform poorly with these data, which makes better suited for the task at hand than most empirical data sets.

We start by analyzing the distribution of Euclidean norms of data points thus generated to illustrate relevant artifacts characteristic of high-dimensional data spaces to show how the growing dimensionality affects performance of our data structure.

4.1 Uniform Data Distribution

We start our data analysis by computing statistical moments for the Euclidean norm of data points. We calculate the minimum, the maximum, the average, the median and the standard deviation for data sets of dimension 10, 20, 40, 60, 80, 100, 150 and 256. Figure 6 shows the results we got. As we can see, the average and the median have the same value for all dimensions, meaning that we have a normal distribution for the values of the Euclidean norm. We can also observe that the standard deviation as a value near zero and that it remains the same for all dimensions. The amplitude

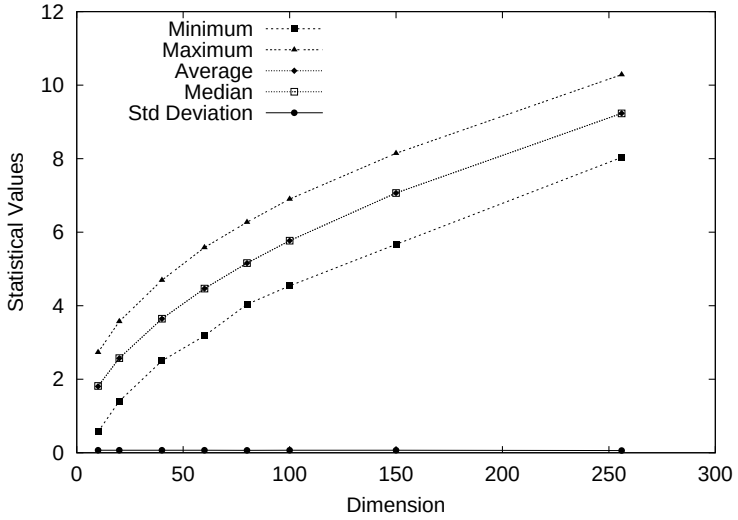


Figure 6: Statistical values for the Euclidean norm.

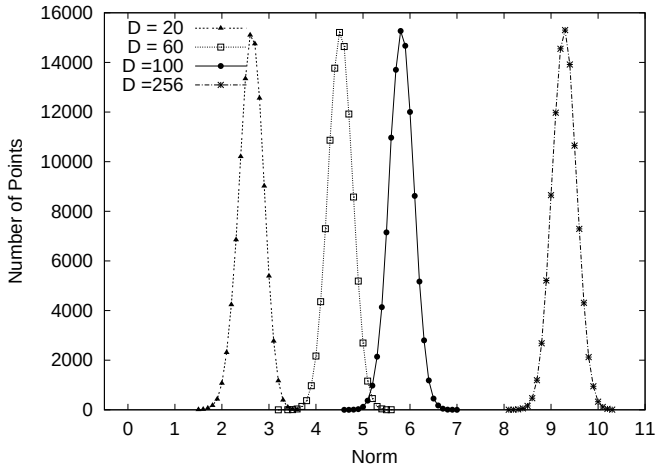


Figure 7: Distribution of norms for several dimensions.

(maximum - minimum) of the norm values is always constant for all dimensions. Finally, we can see that the minimum, the maximum, the average and the median present the same evolution along the dimension.

Our next step was to study the distribution of points according to the Euclidean norm. To that end we compute the mode of the norm for each dimension, using intervals of 0.1. Figure 7 presents the point distribution for dimensions 20, 60, 100 and 256. As we can see the points have the same distribution (normal) for all dimensions.

4.2 Other Distributions

Besides the uniformly distributed data, we also analyzed a data set of real data from the NIST database. Data points have a dimension of 256 and coordinate values can each take the value zero or one. Finally, to simulate an application domain where the data points have different dimensions, we generated synthetic data with variable dimensions and analyzed it.

The main objective of this study was to achieve some conclusions about behavior of the Euclidean norm when we move from uniformly distributed data to real data and see what kind of data sets yield worst performance for the NB-Tree approach. To that end, we computed the distribution of points per norm interval and present it in Figure 8.

As we can see, for uniformly distributed data the values of the norm form a "tight" cluster around a single value. In the case of the real data and of points with variable dimensions, the values of the norm are more evenly distributed. This wide distribution of the norm values seems to validate the NB-Tree approach of using the Euclidean norm as a method to index high-dimensional

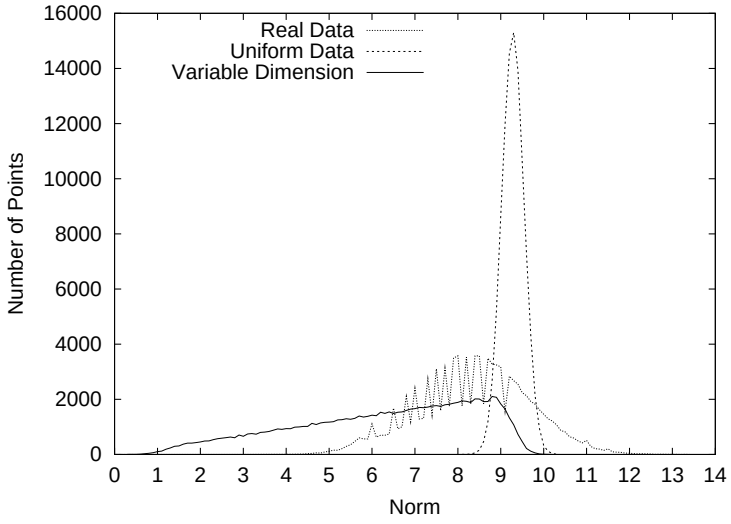


Figure 8: Distribution of norms for different types of data sets.

real data sets of fixed or variable dimensions.

5 Experimental Evaluation

While our approach seems to yield commensurable times to other structures tested, we had difficulties comparing it to other approaches, since some of them crashed on data sets of significant size, preventing comparison. We chose the SR-Tree, the A-Tree and the Pyramid Technique as benchmarks because they are the more recent indexing structures and because there are reliable and stable implementations, which provide correct results and scale up to our intended test data sizes.

In this section we describe the experimental evaluation performed to compare our NB-Tree with the SR-Tree, the A-Tree and the Pyramid Technique. We conducted a set of experiments to analyze final tree size (on disk), creation times and query times as a function of data set dimension and size. All experiments were performed on a PC Pentium II @ 233 MHz running Linux 2.4.8, with 384 MB of RAM and 15GB of disk. Our NB-Tree algorithms were developed using the *dbm* implementation of the B^+-Tree.

First, we present the results of the evaluation for several data sets of uniformly distributed data. Then, we introduce our observations for some real data sets and for a data set of variable dimension.

We did not measure I/O accesses for four reasons: First, nowadays computers can handle databases of millions of high–dimensional points in main memory. Second, after a dozen queries all the indexing structure is loaded into main memory. Third, users of content-based retrieval systems

do not care about the number of I/O operations, but they do care about response time. Fourth, in many indexing schemes, data structure overhead dominates I/O, especially after most of the database is memory–resident. Finally, we assume that locality of reference holds for well–behaved features.

Thus, during our experimental evaluation we start by loading all the structures into main memory and after that we measure search times. We think that the time taken to load the NB-Tree into main memory is not significant, taking into account that it is done just one time. The NB-Tree takes less than one minute to load, for dimensions up to 100 and less than 4 minutes for dimensions up to 256.

5.1 Performance Measures with Uniformly Distributed Data

We evaluated the structures using data sets of randomly generated uniformly distributed data points of fixed size (100,000) and various dimensions (10, 20, 30, 40, 60, 80, 100, 150 and 256). We also created data sets with fixed dimension (20) and various sizes (250,000, 500,000, 750,000 and 1,000,000). Additionally, we randomly generated a set of 100 queries for each dimension, which we later used to evaluate the searching performance of each approach.

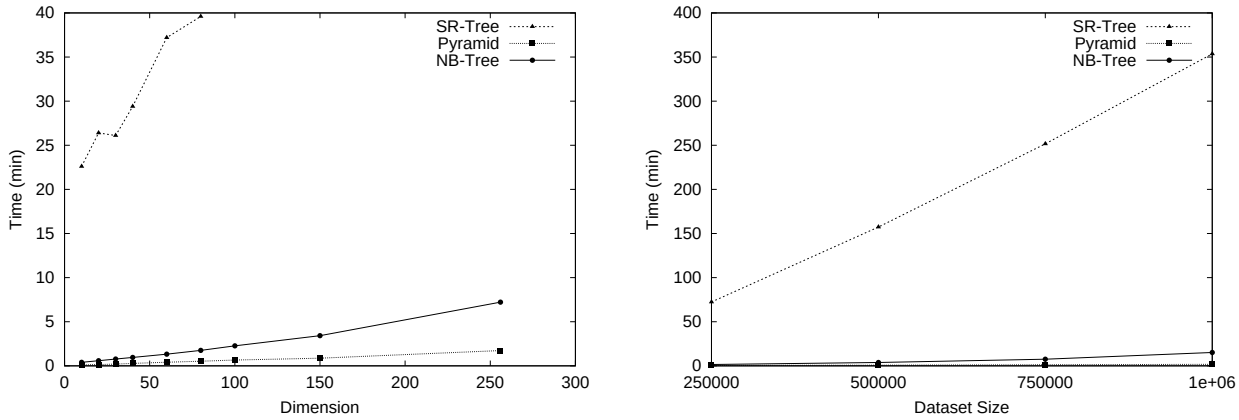
We selected the number of nearest neighbors to search for to be always ten, while the radius (width) of the ball (rectangle) was different for each data set depending on the dimension of the data points. This variation in radius is due to increasing average distance between points when the dimension increases. We found the typical distance between data points to grow with the logarithm of the (vector space) dimension for uniformly distributed data.

In the next subsections we present the results of our experimental evaluation organized by creation, searching and final tree size.

5.1.1 Comparing Creation Times

Most published work tends to ignore tree insertion times. This is because conventional scenarios focus on large static databases which are far more often queried upon than updated. However, there are many applications requiring frequent updating of data sets. For these applications, low insertion times are an important usability factor.

We have compared the creation times to these of the SR-Tree, A-Tree and Pyramid Technique. Figure 9.a shows the time spent to create each structure when the dimension of the data changes. We do not present the creation times for the A-Tree because they are very high, even for low



a) Data set size of 100,000 points

b) Data points of dimension 20

Figure 9: Creation times. a) as a function of dimension. b) as a function of data set size.

dimensions (around 300 minutes for dimensions 10 and 20). As we can see, the NB-Tree and the Pyramid Technique largely outperform the SR-Tree. While the NB-Tree takes 24 seconds to insert 100,000 points of dimension 10, the SR-Tree takes 23 minutes. If we now consider higher dimensions, such as 80, the difference increases even more with the NB-Tree taking 2 minutes and the SR-Tree taking 40 minutes. The Pyramid Technique reveals creation times below the two minutes, for dimensions up to 256.

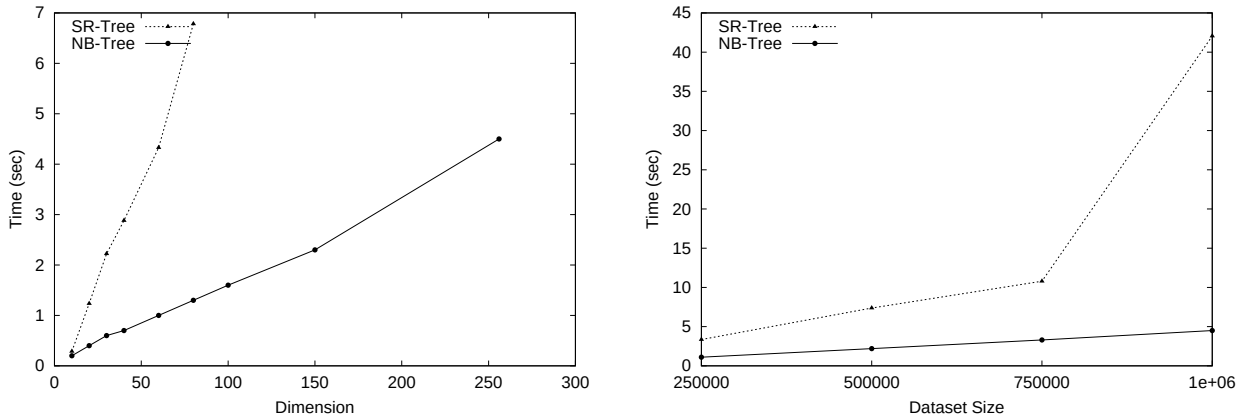
Figure 9.b shows insertion times for data sets of varying size. Although, all structures seem to exhibit linear growth with dimension, SR-Tree creation times grow faster than those of the NB-Tree and of the Pyramid Technique. While the NB-Tree requires no more than 15 minutes to create a tree with one million data points, the SR-Tree takes around six hours and the Pyramid Technique takes only two minutes. From this observation it is clear that the Pyramid Technique and the NB-Tree are more suited to large data sets than the SR-Tree.

We were not able to create the SR-Tree for data sets of dimension bigger than 100, in our system. Thus, in the following subsections we do not display values for dimensions higher than 100 corresponding to the SR-Tree.

5.1.2 Comparing Range Query Times

Another useful primitive in multidimensional data is the range query, where we want to look for data points within a specified distance of the target.

Figure 10.a shows performance for both trees as a function of the dimension. Once again the NB-Tree largely outperforms the SR-Tree for all dimensions. While our approach exhibits searching



a) Data set size of 100,000 points

b) Data points of dimension 20

Figure 10: Range Query searching times. a) as a function of dimension. b) as a function of data set size.

times smaller than one second for dimensions up to 60, and smaller than 2 seconds for dimensions up to 100, the SR-Tree needs more than six seconds. Additionally we can see that the NB-Tree has a linear growth with the dimension while the SR-Tree seems to grow quadratically.

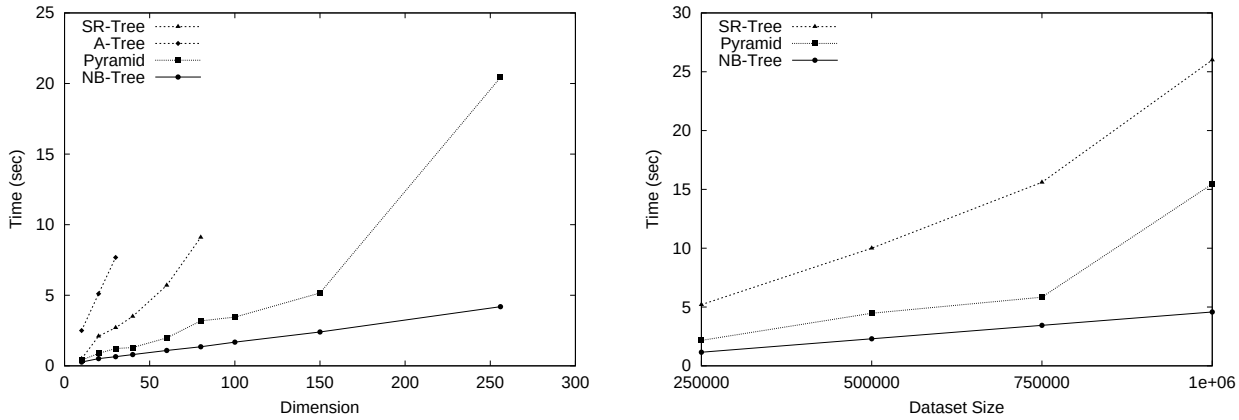
Figure 10.b plots times required for range queries with varying data set sizes. The NB-Tree outperforms the SR-Tree for all data set sizes. The NB-Tree takes only five seconds to perform a range query in a 1 million point data set, while the SR-Tree spends around 40 minutes to yield the same results.

We do not present results for the A-Tree, because the available implementation of the A-Tree does not support range queries. While we have tried hard to evaluate the range query algorithm of the Pyramid Technique we have been unable to reproduce results using the currently available implementation.

5.1.3 Comparing k-NN Query Times

Nearest-neighbor queries are useful when we want to look at the point in the data set which most closely matches the query.

Figure 11.a depicts the performance of nearest neighbor searches when data dimension increases. We can see that the NB-Tree outperforms all the structures evaluated, for any characteristic dimension of the data set. Our approach computes the ten nearest neighbors in less than one second for dimensions up to 40, less than two seconds for dimensions up to 100 and less than five seconds for dimensions up to 256. Moreover, we can notice that the NB-Tree shows linear behavior with



a) Data set size of 100,000 points

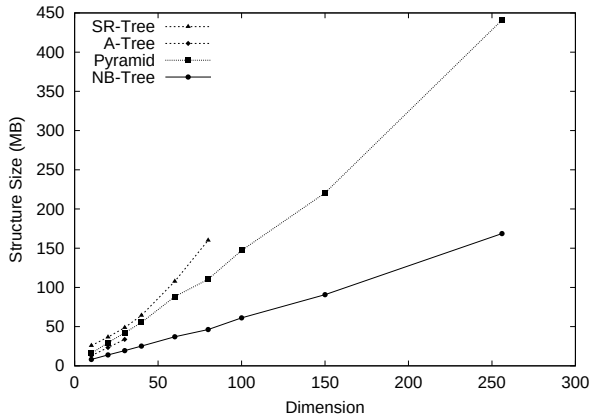
b) Data points of dimension 20

Figure 11: Search times for k-NN. a) as a function of dimension. b) as a function of data set size.

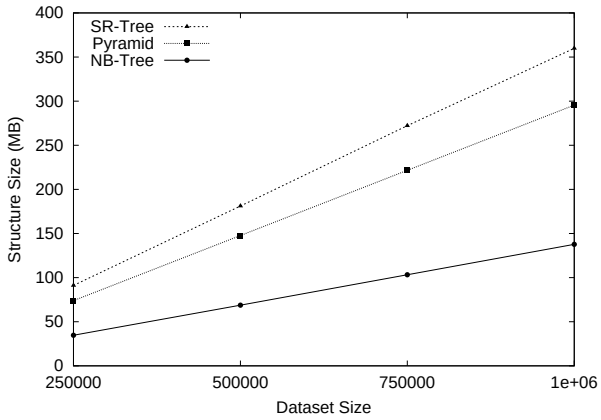
the dimension while the SR-tree and the A-Tree seems to exhibit at least a quadratic growth or worse. The Pyramid Technique performs better than the A-Tree and the SR-Tree, but it shows worst search times than the NB-Tree, for all dimensions. Figure 11.b also shows that the NB-Tree outperforms all structures for k-NN queries in terms of resources required.

5.1.4 Storage Requirements

Looking at Figure 12 we can realize that the NB-Tree requires less storage space than any other structure. The SR-Tree uses at least three times more storage space than the NB-Tree, while the Pyramid Technique requires more than twice the storage space used by our structure. The A-Tree seems to use a storage space of the same size of the Pyramid Technique. Furthermore, the SR-Tree size seems to increase quadratically with the data dimension while the NB-Tree and the Pyramid Technique grow linearly. This linear growth is inherited from the B^+-Tree. We can also see that all structures present a linear growth with data set size (c.f. Figure 12.b). However, the storage requirements from our approach grow slower than those from the SR-Tree and the Pyramid Technique. Even though, the NB-Tree and the Pyramid Technique share a B^+-Tree to store information and even though they store the same information (high–dimensional point + 1D value) the files produced by the Pyramid Technique are around twice the size of the ones from the NB-Tree. The only explanation that we encounter is the different implementations used. We used the `dbm` version, while the Pyramid Technique uses their own.



a) Data set size of 100,000 points



b) Data points of dimension 20

Figure 12: Final tree size. a) as a function of dimension. b) as a function of data set size.

5.2 Performance Measures with Variable Dimension Data Sets

After our experimental evaluation with synthetic uniformly distributed data sets, we tested the indexing structures with other synthetic data set that try to simulate data points of variable dimensions. To that end, we filled the first N coordinates of points with values and the rest $(D - N)$ with zeros. We generate points with dimensions between 4 and 256 (but all points have 256 coordinates). We had to do this, because the other structures do not support points of different dimensions at the same time.

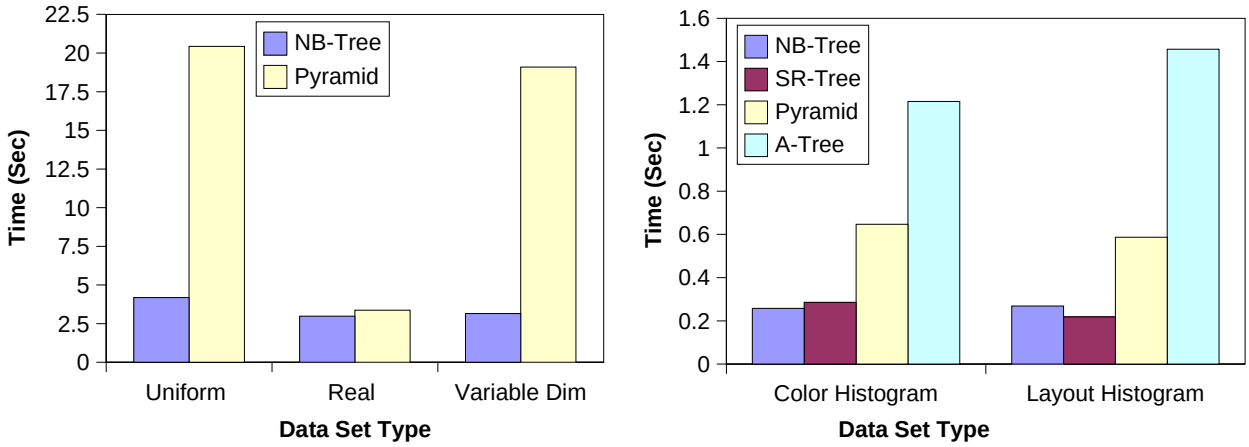
From Figure 13.a(right columns) we can see that the NB-Tree is six times faster than the Pyramid Technique finding the ten nearest neighbors. We do not present values for the A-Tree and the SR-Tree, because these structures can not handle data sets of this dimension (256).

5.3 Performance Measures with Empirical Data Sets

Finally, we evaluate the indexing structures using three real data sets. Two of them had dimension 32 and contain image features extracted from Corel image collection. One contains 68,000 color histograms, while the other contains 66,000 color histogram layouts. The last real data set has 100,000 points of dimension 256, each one representing an instance of a hand-drawn digit. This data set is part of the NIST database.

Due to space limitations, we just present experimental values for the k-NN search.

From Figure 13.a we can see that for any kind of data set with dimension 256, the NB-Tree is always faster than the Pyramid Technique and that the NB-Tree takes less time for the real data set.



a) Data sets of dimension 256

b) Data sets of dimension 32

Figure 13: k-NN searching times for real data sets.

Figure 13.b presents times for the two real data sets of dimension 32. As we can see the NB-Tree and the SR-Tree have the best performance, with query times around 270 milliseconds. The A-Tree once again presents the worst results, taking five times longer than our approach.

Contrary to published [15] experimental values, we have found better query times for the SR-Tree over the A-Tree. Maybe a fine tuning would yield a different outcome. This did not succeed for the many combinations we have tried.

6 Conclusions

In this paper we proposed a simple, yet efficient structure for data of highly–variable dimensionality, the NB-Tree, using the Euclidean norm to reduce dimension and a B^+-Tree to store the 1D values. We described simple algorithms for insertion, point, range and nearest neighbor queries.

We have conducting experimental evaluations using either uniformly distributed data sets, variable dimension and real data. Results show our method to outperform the Pyramid Technique[2], the SR-Tree and the A-Tree. Difference in performance is expected to increase as data dimensionality and data set size increase, and for real (skewed) data points.

Our experimental evaluation indicates that the NB-Tree can efficiently support a wide type of queries, including point, range and the most used (at least in multimedia databases) nearest neighbor queries. This is significant because most indexing structures proposed for range queries are not designed to efficiently support similarity search queries (KNN) and metric-based approaches

[2]KNN queries only. We were not able to reproduce results in range queries.

proposed for similarity queries are considerably more difficult to apply to range queries. In contrast to other approaches which use complex algorithms (combination of MBRs and MBSs to describe partitions, division of the space using hyper-pyramids, etc.) ours relies on very simple (and therefore practical) methods.

Tree-based techniques do not seem to scale up well with growing data set sizes typical of multimedia databases, where it is relatively easy to assemble large collections of documents. The overhead introduced by data-structure traversal and book keeping in these approaches far outweighs the potential advantages of sophisticated spatial indexing structures.

For a seemingly "bad" point set, generated from uniformly distributed data, both the range and k-NN search algorithms can degenerate into a sequential search, where we have to compute the distance from the query point to all the points in the data set. However, as we have seen in section 4 real data and data of variable dimension both exhibit Euclidean norm distributions that better match our selection and indexing methods. We conjecture that our approach tends to excel for sparsely–distributed data where clusters tend to be evenly distributed throughout the hyperspace. Moreover, since our algorithms are very simple, response times remain reasonable for data sets of considerable size and dimension

We are now using the NB-Tree in a trainable shape recognizer [7, 6], as KNN mechanism. Results so far show that query times are very low, making it a good choice for interactive applications where immediate feedback is required.

In summary our work presents a simple method which is able to handle data of variable dimensions in an efficient and scalable manner suitable for interactive use.

Acknowledgments

This work was funded in part by the Portuguese Foundation for Science and Technology, project 34672/99 and the European Commission, project SmartSketches IST-2000-28169.

References

[1] Stefan Berchtold, Christian Bohm, H. V. Jagadish, Hans-Peter Kriegel, and Jorg Sander. Independent quantization: An index compression technique for high-dimensional data spaces. In *Proceedings of the 16th International Conference on Data Engineering (ICDE'00)*, pages 577–588, San Diego, USA, 2000.

[2] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegel. The Pyramid-Technique: Towards Breaking the Curse of Dimensionality. In *Proceedings of the International Conference on Management of Data (SIGMOD'98)*. ACM Press, 1998.

[3] Stefan Berchtold, Daniel Keim, Hans-Peter Kriegel, and Thomas Seidl. Indexing the solution space: A new technique for nearest neighbor search in high-dimensional space. *IEEE Transactions on Knowledge and Data Engineering*, 12(1):45–57, 2000.

[4] Stefan Berchtold, Daniel A. Keim, and Hans-Peter Kriegel. The X-tree: An Index Structure for High-Dimensional Data. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB'96)*, Mumbai(Bombay), India, 1996.

[5] Kaushik Chakrabarti and Sharad Mehrotra. The Hybrid Tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering (ICDE'99)*, pages 440–447, 1999.

[6] Manuel J. Fonseca and Joaquim A. Jorge. CALI : A Software Library for Calligraphic Interfaces. INESC-ID, available at `http://immi.inesc-id.pt/cali/`, 2000.

[7] Manuel J. Fonseca and Joaquim A. Jorge. Experimental Evaluation of an on-line Scribble Recognizer. *Pattern Recognition Letters*, 22(12):1311–1319, 2001.

[8] Manuel J. Fonseca and Joaquim A. Jorge. Towards content-based retrieval of technical drawings through high-dimensional indexing. *Computer and Graphics*, To appear, 2002.

[9] A. Henrich. The LSDh-tree: An access structure for feature vectors. In *Proceedings of the 14th International Conference on Data Engineering (ICDE'98)*, pages 362–369, 1998.

[10] A. Henrich, H.-W. Six, and P. Widmayer. The LSD tree: Spatial access to multidimensional point and non-point objects. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB'89)*, pages 45–53, 1989.

[11] N. Katayama and S. Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. In *Proceedings of the International Conference on Management of Data (SIGMOD'97)*, pages 369–380. ACM Press, 1997.

[12] K.-I. Lin, H. Jagadashi, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–543, 1994.

[13] D. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing mechanism with good garanteed performance. *ACM Transactions on Database Systems*, 15(4), 1990.

[14] B. C. Ooi, R. Sacks-Davis, and K. J. McDonell. Spatial K-D-tree: An indexing mechanism for spatial databases. In *Proceedings of IEEE COMPSAC Conf.*, 1987.

[15] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, and Haruhiko Kojima. The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB'00)*, pages 516–526, Cairo, Egypt, 2000.

[16] J. Shepherd, X. Zhu, and N. Megiddo. A fast indexing method for multidimensional nearest neighbor search. In *SPIE Conference on Storage and Retrieval for Image and Video Databases*, pages 350–355, 1999.

[17] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB'98)*, pages 194–205, New York, USA, 1998.

[18] David A. White and Ramesh Jain. Similarity Indexing: Algorithms and Performance. In *Proceedings SPIE Storage and Retrieval for Image and Video Databases*, 1996.

[19] David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th IEEE International Conference on Data Engineering*, pages 516–523, 1996.

[20] Cui Yu, Stéphane Bressan, Beng Chin Ooi, and Kian-Lee Tan. Querying high-dimensional data in single dimensional space. *VLDB Lournal*, To Appear, 2002.