Abstract

The importance of multimedia databases has been growing over the last years in the most diverse areas of application, such as: Medicine, Geography, etc. With the growth of importance and of use, including the explosive increase of multimedia data on the Internet, comes the larger dimensions of these databases. This evolution creates the need for more efficient indexing structures in a way that databases can be useful, returning accurate results in a short time. Typically, these databases use multi-dimensional indexing structures to deal with feature vectors extracted from multimedia elements. However, the majority of existing multidimensional indexing structures, suffer from the well-known “dimensionality curse”, making the search in high-dimensional spaces a hard problem. In this work we developed an efficient indexing structure to support large databases containing data of high dimensions (over 100). The new indexing structure, ND-Tree (Norm Diagonal Tree), is based on a new dimension reduction technique based on two metric measures, Euclidean norm and distance to the unity cube diagonal. Experimental results show that our solution is more efficient and that the two metric measures provide a more effective pruning mechanism, while compared to other indexing structures.

Keywords

Multidimensional indexing structures, dimension reduction, dual metric-system.

1. Introduction

Typically multimedia databases use multi-dimensional descriptors, vectors containing a set of features extracted from multimedia objects, as a way to represent them. However, as time goes by, it is possible to extract a large number of features from objects, since we have more efficient ways to extract information and also because the new multi-dimensional indexing structures support descriptors of higher dimensions. These features are used to compare and identify objects in a more efficient way, since the comparison is just the computation of a distance between feature vectors. Moreover, the growth of dimensions, also imply the enlargement of the databases size.

With the growth of the database size and of the vectors dimension it is important to find new and more efficient ways to index and search for multimedia objects (i.e. feature vectors).

However, searching in high multi-dimensional spaces, usually, is not very efficient due to the “dimensionality curse”.

To solve this, some solutions were developed, being the most important aspect of them the data space partitioning. The better we partition the space, the more efficient and the better the searching results, like k-NN search or window search will be. The k-NN search is a search that returns the k nearest neighbors to a given point. These nearest neighbors are the most similar data to the one used as query in the search.

There are already a large number of solutions in this area, that can be grouped in four distinct categories: the first category contains the techniques based on data and space partitioning; the second category contains the techniques based on the representation of the original feature vectors, using smaller approximate representations; the third category consists in techniques that use metric-based methods as an alternative direction for the high-dimension indexing; and the fourth and final category is constituted by the transformation-based indexing schemes. Despite all these techniques, one thing is common to almost all of them, if not to all, the “dimensionality curse”, where performance drastically decreases when the dimension increases.

The main goal of our work is to create a multidimensional indexing structure and correspondent searching methods that will outperform the best techniques that already exist. We want our technique to be faster, while returning results for a query, and to have a good pruning algorithm, comparing fewer values to achieve the final result. We also want to support dimensions larger than 100, to work with large databases, to support dynamic insertion and removal and to work well with any kind of point distribution (e.g. clustered and uniformly distributed).

When we say for instance that we want to support dimensions larger than 100, we must not forget that we also want to work well with low dimensions, trying to achieve at least a linear performance, whatever the scenario is.

Another important objective is that our k-NN search should return exact results and not approximated results, as some existing solutions do.

The solution achieved is based on dimension reduction using a dual metric system, the Euclidean norm and the distance to the main diagonal of the unity cube. The main diagonal is the diagonal that goes from the origin of the hyper-cube to the opposite end. With these two metrics we can optimize the pruning
process during search, reducing the number of comparisons, and the computation time.

After the dimension reduction to 2D, points are kept in a Quad-Tree like structure to optimize the search in the low dimension space. Then, when we want to do any query, searches are performed in the 2D space, speeding up the process.

The major contribution of our work is the new dual metric technique. This, in combination with a Quad-Tree structure, demonstrated that we can achieve better results than with the techniques we compared. We achieved better times and compare less points to get exact results, for different types of queries.

The rest of the dissertation is organized as follow. We survey the related work in Section 2 and take conclusions about them. In section 3 we explain in detail the overview of our solution and some detail of the architecture of our solution. In Section 4 we present the characterization of the data sets we used on our tests. Section 5 presents and analyzes the experimental results. We conclude this report with our conclusions, Future Work and an analysis of the computational complexity of the solution in Section 6.

2. Related Work

The research for addressing the high-dimensional indexing problem has been always very active, and has been around for a long time. Techniques presented so far can be divided into four main categories [1]. Techniques based on data and space partitioning; techniques based on the representation of the original feature vectors using smaller approximate representations; techniques that use a metric-based method [12] has an alternative direction for the high-dimension indexing and the last one is constituted by the transformation-based indexing schemes.

The first category contains techniques based on data and space partitioning, like R-Tree [2], PR-Tree [8], etc. R-Tree was one of the first techniques being the other techniques variants. These methods, in general, work well at low dimensionality (for instance, the R-Tree work well for dimensions between 2-10), but their performance deteriorates fast as the dimensionality increases, due to the famous "dimensionality curse".

An R-Tree[2] is a height-balanced tree similar to a B-Tree, containing pointers to data objects on leaf nodes. It splits space with hierarchically nested, and possibly overlapping, minimum bounding rectangles (MBR). It uses a dynamic index, allowing inserts, deletes and searches, with no need for a periodic reorganization. The search algorithm is similar to the one used on the B-Tree, however more than one sub-tree under a visited node, may need to be searched since it is not possible to guarantee good worst-case performance.

R-Trees are known to have problem with the increasing dimension of the data, and are also inefficient in supporting range queries in high-dimensional databases. Another huge problem with R-trees and some of the variants is the overlap of bounding boxes in the directory, which increases when the dimension grows.

The PR-Tree[8], or Priority R-Tree, was the first variant of the R-Tree[2] that always answers a window query using the same asymptotic time, depending on the number of the hyper-rectangles stored in the R-Tree, the disk block size and the output size. The PR-Tree uses a k-dimensional pseudo-PR-Tree to answer window queries efficiently, but the pseudo-PR-Tree is not a real R-Tree, since it does not have all the leaves at the same level. For that, the real PR-Tree is obtained from the pseudo-PR-Tree.

The general problem that we can clearly identify in this first category is that as the number of dimensions increases, we cannot partition the space along each dimension, or else we get empty pages. The queries then degenerate and intersect almost all pages, turning the queries into a sequential scan.

The second category contains the techniques based on the representation of the original feature vectors, using smaller approximate representations. Among these techniques we can find the VA-File [9], A-Tree [11], etc.

The authors of the VA-File established lower bounds on the average performance of existing partitioning and clustering techniques, demonstrating that methods such as R*-Tree[3], the X-Tree[5] and the SR-Tree[7], which are methods that are known to suffer from the dimensionality curse, are outperformed by a sequential scan whenever the dimensionality is above 10. The VA-File is based on the idea of object approximation, as it has been used in many different areas of computer science. Examples are the Multi-Step approach of Brinkhoff et al.[26], which approximates object shapes by their minimum bounding box, the signature-file for partial match queries in text documents, and multi-key hashing schemes. In the comparison of results with other methods derived from the R-Tree, the VA-File proves to be better in terms of performance, but it was only tested for 5 and 6 dimensions.

Although the VA-File was intended to solve the dimensionality curse that could not be resolved by the R-Tree Variants, it still deals with a minor number of dimensions of those that are used nowadays. As we can see, tests performed by the authors of the VA-File, were all with a low number of dimensions (6 to 45 dimensions).

One of the great disadvantages of this method is that the performance depends heavily on the ordering of vectors. Other weaknesses that can be identified are: the construction time, the distance ratio decreases with increase of dimensions, the percentage of found $k$ nearest points is usually low, search time for low dimensions like 1,2 or 3 is larger than the quad-tree or the R*-Tree.

The basic idea behind the A-Tree[11] (Approximation Tree) is the introduction of the Virtual Bounding Rectangles (VBRs), which contain and approximate MBRs and data objects. VBRs can be represented rather compactly, and thus affect the tree configuration both quantitatively and qualitatively.

The A-Tree has been experimented against the SR-Tree[7] and the VA-File[9] using both synthetic and real data sets. For the real data sets, the A-Tree outperforms the SR-Tree and the VA-File in all range of dimensionality up to 64 dimensions, and it also saves 77.3% in page accesses compared to the SR-Tree and 77.7% compared to the VA-File, for 64-dimensional real data. The $k$-nearest neighbor search algorithm for the A-Tree is an improvement on the algorithm presented in [21].
One of the problems of the A-Tree, is the insertion time, which is too big, being bigger than the insertion time of the SR-Tree, especially if the full utilization feature is used. The full utilization is only recommended for use in a static data set. In [18] experimental tests performed by the authors confirm the big insertion times of the A-Tree, which is not a good result for what we want to achieve. Those results show also that the A-Tree is not prepared to support the dimensions we want to work with, that will be always larger than 100.

In the techniques presented in the second group we can identify a very important problem. They use an approximated vector, which means the results of the queries are not exact as we want ours to be. Other problem is the low dimensions (between 4 and 64) they support.

The third category consists of techniques that use metric-based methods [12] has an alternative direction for the high-dimension indexing. Among those techniques we can find for example, MVP-Tree [13], Slim-Tree [15] etc.

The MVP-Tree is a method that introduces a distance based index structure called multi-vantage point (MVP). The MVP-Tree uses more than one vantage point to partition the space into spherical cuts at each level. It also utilizes the pre-computed (at construction time) distances between the data points and the vantage points.

One of the problems we can identify in the MVP-Tree is that it is a static index structure, which means that once the structure is in use, there cannot be any inserts or deletes to the points present in that structure. The problem is that the MVP-Tree is a distance-based index in a way that the authors concluded that there wasn’t a cost efficient solution to impose a global total order or a grouping mechanism on the objects of the application data domain.

The Slim-Tree [15] is a dynamic metric access method (MAM), which shares its basic data structure with other trees like the M-Tree[14] where data is stored in the leaves and an appropriate cluster hierarchy is built on top.

The algorithm used by the Slim-Tree was derived from the author’s findings that high overlap in a metric tree is largely responsible for its inefficiency. Unfortunately, the well-known techniques to measure overlap of a pair of intersecting nodes (e.g. circles in a two-dimensional space) could not be used for metric data.

Although the Slim-Tree helps to solve the overlap problem of the M-Tree, its Slim-Down algorithm is just too expensive.

The fourth and final category is constituted by the transformation-based high-dimensional indexing schemes. Among those techniques we find techniques such as Pyramid Technique [17], NB-Tree [18], iDistance [19], etc.

The Pyramid-Technique is based on a special partitioning strategy which is optimized for high-dimensional data. The basic idea is to divide the data space first into 2d pyramids sharing the center point of the space as a top. In a second step, the single pyramids are cut into slices parallel to the basis of the pyramid. Furthermore, this partition provides a mapping from the given d-dimensional space to a 1-dimensional space, making the use a B+-Tree to manage the transformed data possible.

The Pyramid-Technique [17] is efficient for window queries, but not for k-NN queries which is our objective. If the Pyramid-Technique is used for a k-NN query, it is very likely that the query degenerates into a sequential scan, because it will catch most of the pyramid zones.

The NB-Tree [18] is an indexing technique based on a simple, yet efficient algorithm to search points in high-dimensional spaces, using dimension reduction. Multidimensional points are mapped to a 1D line by computing their Euclidean norm. In a second step these are sorted using a B+-Tree on which all the subsequent operations are performed.

This technique provides a simple and compact means to indexing high-dimensional data points. Authors use an efficient 1 dimension data structure, the B+-Tree to index points sorted by their Euclidean norm. This way, all the operations are performed on the B+-Tree. Since this is the most efficient 1-dimensional structure, the NB-Tree inherits its good performance, specifically for point queries. To create an NB-Tree we start by computing the Euclidean norm of each N-dimensional point from the dataset. The resulting norm and the N-dimensional point are then inserted in a B+-Tree, using the norm as key. After insertion of all points we get a set of N-dimensional points ordered by their norm value.

The k-NN search starts by doing a ball query. After this ball query, it checks if it has enough points inside the ball that can satisfy the query. If not, it starts an iterative process, where the size of the ball increases gradually until it gets all the points specified by the query (view Figure 4).

The drawback of NB-Tree is that it cannot significantly prune the search region; especially when the dimensionality becomes larger, the pruning capability of it can be so poor that the number of candidate points returned by the first round becomes too large to be filtered effectively.

The iDistance [19] was presented as a new technique for k-NN search that can be adapted to different data distributions. In this technique, first the data is partitioned and is defined a referenced point for each partition. Then it indexes the distance of each data point to the reference point of its partition. Since this distance is a simple scalar, with a small mapping effort to keep partitions distinct, a classical B+-Tree can be used to index this distance.

One of the drawbacks of the iDistance is that the query efficiency relies largely on clustering and partitioning the data and is significantly affected if the choice of partition scheme and reference data points is not appropriate. It is also important to note that the results presented by the k-NN search are not exact. It has a dynamic index.

Even with some of the techniques on this group being able to support larger dimensions then the ones in previous sections, they are still too low for what we want. The largest dimension is supported by the NB-Tree which supports from 8 to 256 dimensions. It is also the only technique in this section that has exact results on the K-NN query. These are some of the reasons why we chose the NB-Tree for our comparative tests.

Almost all solutions analyzed use a dynamic structure, which is an essential feature, in our perspective, since the
possibility to insert and remove elements dynamically is an important feature in every application, otherwise it would be necessary to recreate everything each time the database is changed. There are only a small number of techniques which use dimension reduction, which is an area where we will present a different approach from the few available, since we intend to reduce the dimension of the feature vectors to 2D. We think that dimension reduction is one of the best ways towards solving the “dimensionality curse». The most similar approach to ours is the one used by the EHD-Tree, since it also uses a dual metric system, but it has the problem of returning only approximate results in the K-NN queries and the fact that it supports a low number of dimensions. The dual metric system seems to be the best choice in order to achieve a good pruning.

Our objective is to return exact results to the k-NN searches.

It can also be noted, that almost all of the techniques analyzed support less dimensions than the ones we want to support with our solution.

After analyzing these techniques, the NB-Tree clearly becomes the best technique to compare ours, since it is the solution that supports the largest number of dimensions, the results provided are exact and it is also dynamic. By analyzing the results of the experimental tests performed by the authors of the NB-Tree, we can see that it surpasses the majority of the techniques mentioned above. So if we prove that we achieve better results than the NB-Tree, we can conclude that it is highly probable that we have better results than those same techniques.

We intend to overcome the NB-Tree by having a much better pruning using our dual metric system.

3. The ND-Tree

The general idea of our solution is to reduce the dimension of multidimensional points to two dimensions. These 2D points are then inserted in a Quad-Tree structure for further organization and searching.

In more detail, as we receive the high-dimensional feature vectors extracted from multimedia objects, we reduce their dimension to two, using a dual metric approach, as illustrated in Figure 1. One of the metric measures is the Euclidean norm, the same used in the NB-Tree [18]. The second metric used is the distance to the main diagonal of the hyper-cube. This way we intend to have a better pruning approach than we would get by using just one measure.

With this dimension reduction technique we can accept feature vectors with any number of dimensions, since we are always working with two dimensions. The calculations are exactly the same.
In Figure 2, we can see a practical example using a set of points with 2 dimensions. We can see the resulting transformation of our technique in Figure 3. In this example we are doing a range query and we demonstrate which areas would be scanned in the algorithm that we’ll explain further on this dissertation. In more detail we can see that for each point in Figure 2, we calculate its Euclidean Norm and its distance to the main diagonal, which are the two coordinates used in our system that is represented in Figure 3. The divisions presented in Figure 3, are the divisions created by the Quad-Tree structure in order to decide where to insert the points. In this example, we are assuming a Quad-Tree structure that subdivides when the limit of 2 points by area is reached.

3.2 Structure

With our dimension reduction technique the result is always a two dimensional point, being one of the coordinates the distance to the main diagonal and the other the Euclidean norm.

At first we considered the B+-Tree because of the advantages of that structure, such as being dynamic, balanced, and more. But the problem was that the B+-Tree is only useful for 1 dimension, and we are working with 2 dimensions. Because of this, we studied the possibility of converting our 2 dimensions into only 1, but that 1 dimension still had to reflect the distance between the points in an exact way. We couldn’t find a solution to this problem. Even if we used more than one B+-Tree, for example, if we divided the Norm into ranges and within each range we had a B+-Tree with the diagonal value for index, we would still lose accuracy and couldn’t confirm if the results were exact or not.

Given this situation, we had to find a new structure to insert our 2D points.

We studied the kd-Trees but the divisions of the kd-Trees vary according to the points, so it was not a good solution for us, since we wanted to create a dynamic structure. If we used the kd-Trees we would have to recalculate the space divisions each time we inserted or deleted a single point. That would be too much time consuming.

Since we were sure working with two dimensions, we needed to divide the space in a way we could prune it efficiently. We already knew the kd-Trees were not efficient but where on the right track. So we had to choose a similar structure but with the divisions with fixed sizes, that didn’t depend exclusively of one point. Given this we studied the Elias structure [27].

It has fixed divisions, which was something we wanted. But we still had a problem. The distribution of real points and random points is not equal. The real points tend to cluster, so the Elias structure was still not the best.

So we needed a structure with fixed divisions but somehow dynamic. So we decided to use a Quad-Tree like structure.

A Quad-Tree is a tree data structure in which each internal node has up to four children. Quadtrees are most often used to partition a two dimensional space by recursively subdividing it into four quadrants or regions. The regions may be square or rectangular, or may have arbitrary shapes, but in our implementation all the regions have the same shape which is rectangular. All forms of Quadtrees share some common features:

- They decompose space into adaptable cells
- Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits
- The tree directory follows the spatial decomposition of the Quadtree

We start by dividing the 2 dimension space into 4 equal sections: the top-left, top-right, bottom-left and the bottom-right. This division is valid for every other sub-division done to any Quad-Tree area. Then as we insert points, each point is inserted into one of these sections.

There is a maximum number of points defined for each sections. That maximum can be user defined or defined according to the number of points inserted on the first time. Whenever a section reaches its maximum capacity of points, it is further subdivided into new 4 more sections and so forth.

With these divisions being dynamic we can assure that we have a good division of points independently of the distribution of points.

We have chosen a Quad-Tree like structure for our solution because it was the one that best suited our approach.
3.3 Range Query

The Range Query returns a set of results that are inside an hyper-sphere with center in the query and with a specified radius.

With these two values we calculate the margins of the region in the Quad-Tree that we have to search however, this region isn’t a simple circle around the query point, since the hyper-sphere is not a circle when converted to two dimensions, through our reduction system. What we have to do is calculate a hyper-sphere for the number of dimensions we are currently working with.

After we have the boundary points of the Hyper-sphere converted to two dimensions, we create a bounding box over that boundary. We then use that bounding box to select the areas to get the points (see figure 4).

We then get all the points from that areas, and compare each one with the query point. If the distance is less than the radius, then that point is a result and is added to the result list. After comparing all points with the query, the list of results is returned.

An example of this query can be seen in Figure 4.

3.4 K-NN Query

The K-NN Query is the most complicated. Our K-NN is implemented as an incremental Range Query, where the radius increases slowly until we get the \( k \) nearest points.

As the radius increases, we keep getting newer points. This points need to be tested in order to decide if they are nearer than the points we had from previous iterations. We keep increasing the radius and testing points until we reach the number of points we had before may be at a larger distance than the ones we are going to get in the next iteration. This happens because of the shape of the regions and the way we chose those regions.

In more detail: in order to decide which the nearest points are, we use an ordered insert, ordered by distance. This way if we have the number of points we need, but the next point is closer than other on the list, we need to keep the new point in order to have the right results.

We keep increasing the radius until we fulfill the stop condition of the algorithm which is: the list must be full and the last point must be inside the break-off radius.

The break-off radius works as a threshold. If the point with the largest distance to the query point is not inside the break-off radius, then we can’t be sure if there are or not other points that are closer to the query and so, we have to keep looking. Only when the last point is inside the break-off radius we can be sure.

The break-off radius is calculated using the 2D bounding box in the Quad-Tree with the following formula:

\[
\text{breakoff} = \frac{\text{BoundingBoxMin} - \text{BoundingBoxMax}}{2}
\]

In the bounding box, the difference between the maximum and the minimum X is always bigger than the difference between the maximum and the minimum Y. That is why we use the X.

We have to do this verification, because we may have filled the line with points that are not the closest to the query. That happens because with the bounding box we don’t get the points that are inside of the box. We get the points that are inside of the regions that are overapped by the box. This way the break-off radius grows as the bounding box grows, and we can guarantee that the results we have are really the closest points to the query (see Figure 4).

As the bounding box grows gradually, we only compare the boundaries with the regions that were not included before. This way we save comparisons.

Once the conditions are fulfilled, the algorithm stops and returns the results.

The insertion ordered array is the list where we temporarily store the points that later will be returned as the results of the K-NN Query. After each insertion we run the QuickSort algorithm in order to keep the list ordered. This was a fast way to get it done, but this could be one of the things for future work, since there are two places where we spend time on the queries: the calculations of the distance between points and the insertion of the points in the list.

Since the K-NN Query is an incremental range query, we can see Figure 4 has one step of its steps.

4. Data Set Characterization

In this section we present an analysis of the different test files used in our tests. The distribution of the points is important to the algorithm results. That is why we use both random and real points since their distributions are different. This way we can approach both sides of the problem.

4.1 Random Points with different dimensions

These random points were generated using the random function of Java for each of their coordinates. What we can verify is that as the dimensions grow, the points get more clustered in our reduced 2D space, as we can see in figure 5.
The fact that the test files are so clustered is the reason why the number of points compared is so high for both structures. These test files are all composed by one hundred thousand points.

The objective of these tests is to analyze the ability of both structures to handle in an effective way the different dimensions of the data points.

4.2 Random Points with different sizes

In this section we present the point distribution of the test files created to evaluate the performance of both structures with a large number of points. We created two sets, one with 20 dimensions, and the other with 50 dimensions. For each of the sets we created five test files with respectively 100,000 points, 200,000 points, 500,000 points, 750,000 points and 1,000,000 points.

The points in these test files, as in the test files presented in the previous section, are also clustered, as we can see in figure 6. This doesn’t affect the outcome of the tests since the objective is to evaluate the performance with a large number of points, whatever their distribution is.

4.3 Real Data Points

In this section we show the distribution of points taken from real data, in opposition to random generated points used in the prior sections. For these tests we used two test files. One is composed by data points that describe the topology of clip-art drawings, while the other is composed by points that describe the geometry of the visual elements in clip-arts. The topology file has 20,007 points with a maximum dimension of 327, while the geometry file has 63,400 points with a maximum dimension of 11.

We state the maximum dimension of each file because not all points have the same dimension but in the ND-Tree structure, we convert all points to the maximum dimension.

We can also see the differences to the randomly generated points. Here the points do not assume that so clustered look.

5. Experimental Evaluation

To evaluate our solution we measured the time needed for insertion, reading and for the different types of queries. We also count the number of points that are compared during the execution of queries. We used data sets randomly generated and from real data points, with different dimensions and sizes. We measured these values for our solution, ND-Tree and for the NB-Tree. For the Range queries we also presented the values for the linear search. We did not present the results of the comparison of the linear search to both techniques on the KNN queries, because the values of the linear search on the KNN are always much larger than both techniques, and to present the three of them in the same graph is to devalue the difference between the ND-Tree and the NB-Tree.

With these tests we intended to prove that the dual metric system we chose can present better results than the single metric system used by the NB-Tree.

We separated the tests in three phases. First we evaluate the differences between the two structures by using randomly generated test files that differ only in dimension. All of them have 100,000 points but vary in dimension so we can evaluate the growth in time implied by the growth in dimension. Secondly we
evaluate the differences using a steady dimension but augmenting the number of points in the files. For these experiments we use two sets of files. One uses 20 dimensions and the other 50. For each of the sets we have files with different sizes, varying from 100,000 to one million points. Finally in the third part of the tests we used real points, in alternative to the randomly generated ones. We did this mainly to assure that our structure wasn’t only better using random points but also using real points since the distribution of points change, affecting the search algorithms.

The NB-Tree used was coded in C++ while our ND-Tree is coded in Java, a factor that must be taken into consideration when evaluating these tests. In the final sub section of this section we show the difference in time between Java and C++ for a function which calculates the distance between two points. This function has been identified as the function where our queries spend the most part of the time. We performed this difference test to show that most of the operations are still faster in C++, which shows that our ND-Tree implemented in C++ would have much better results. This subject will be further explored in the Future Work section.

5.1 K-NN Query Times

5.1.1 Evaluation according to dimension (Random Points)

Figure 8 shows the time spent on the KNN queries, in this case using a $k$ equal to 20. Analyzing the results we can see that the query times for the ND-Tree are better than those ones produced by the NB-Tree for dimensions smaller than 200. The results for dimensions equal or above 200 are better for the NB-Tree, but we believe that this difference can be explained by the difference in speed between Java and C++, since the NB-Tree is implemented in C++ and the ND-Tree is implemented in Java.

Fig. 8 - Time in Seconds spent in a k-NN query using $k = 20$

The graphic in Figure 9 is very similar to the one in the Figure 8, and so the analysis is very similar. The ND-Tree is better until 200 dimensions. After that the NB-Tree shows better results.

5.1.2 Evaluation according to the number of points (Random Points)

Contrary to this same tests performed according to the growth of dimensions, here the query times of the ND-Tree is always faster than the NB-Tree. We can also state that the difference becomes larger has the number of points in the database grow.

The time spent by the NB-Tree performing these queries is roughly the double than the time spent by the ND-Tree performing the exact same queries. Again, the lower the dimension, the larger the difference, which can be explained by the time wasted by the comparison function in Java. More dimensions, more comparisons.
5.1.3 Evaluation using Real Data

For the tests performed to the geometry points, the ND-Tree presents much better results than the NB-Tree, except for the $k$ equal to 5. Apart from that, the rest of the results are much better showing that the ND-Tree is faster performing this queries than the NB-Tree.

5.2 Range Query Times

5.2.2 Evaluation according to dimension (Random Points)

Fig. 14 - Time spent on KNN searches on geometry file

Although the number of points compared for this KNN tests is much larger in the ND-Tree, that doesn’t reflect in the query time, until we perform the KNN query with $k$ equal to 200. For the 200 and 500 neighbours it is reflected in the query time the growth on the number of points to be inserted on the ordered list. This function together with the distance calculating functions are the bottle necks in Java.

Fig. 15 - Time spent on KNN searches on topology file

Fig. 16 - Time in Seconds spent in Ball Query using radius equal to 10% of the maximum norm
As we can see in the graph above, the ND-Tree presents much better results in the range queries. For these tests we chose the radius of the range or ball as percentages of the maximum norm.

We can also see that the times of the ND-Tree keep going up as the dimensions rise, but nothing compared to the time spent by the NB-Tree. As we will be able to see in the next tests, the range queries are a feature where the ND-Tree always presents far better results, for any dimension or even number of points.

Here the times of the NB-Tree are steadier but still above of the ones performed by the ND-Tree. They evolve at the same rate but on totally different scales. The major difference is for the test file with the points of 10 dimensions.

In this test we were not able to achieve acceptable results for the NB-Tree structure when the dimensions were 50 or more, since the system where the tests were executed became unresponsive, becoming impossible to test it on these conditions.

For the results we could compare, the ND-Tree structure was much faster.

These results can be justified by the better pruning system of the ND-Tree, which was one of the main objectives of our work. With it we have less comparisons, since the points are subdivided in more regions than the ones of the NB-Tree. This conclusion can be taken from the next Figure.

5.2.3 Evaluation according to size (Random Points)

Again the difference between the two structures when performing range queries is big. For every test performed in this section we can observe that the bigger the radius used the bigger the difference between the two structures. For the rest of the tests where we used the radius of 25% and 50% the ND-Tree still presents better results. The large value for the 750,000 point file in Figure 20 is due to the distribution of the points in that file which increases the number of comparisons and therefore the result displayed. We can also see that for the ND-Tree that difference isn’t noted as it is in the NB-Tree, which demonstrates
that the ND-Tree is better prepared to deal with different points set configurations.

5.2.4 Evaluation using Real Data

![Fig. 1 - Time spent on range queries on geometry file](image1)

![Fig. 23 - Time spent on range queries on topology file](image2)

The major difference to be pointed out in this range queries is that the NB-Tree performs much better using real data than random data, but still, it is slower than the ND-Tree. This shows that the ND-Tree is all-round faster than the NB-Tree structure.

5.2.5 Java vs. C++

One of the major differences between our structure and the NB-Tree structure used as a comparison measure is that the ND-Tree is coded in Java while the NB-Tree is coded in C++. This has a major impact on the performance, but still we managed to achieve great results.

The impact of Java is noticed in our case as the number of comparisons increased. After a careful analysis, we concluded that the major time spending function of our structure is the comparison function and the ordered insert in the list of results for the queries. Here we will exemplify using only the distance function. This function is used to compare two points, mostly in the range and KNN queries.

With this conclusion we intend to prove that if the ND-Tree structure was to be implemented in C++ the performance would be much better, and with that present better results all round.

![Fig. 24 - Time spent on comparisons in Java and C++](image3)

As we can see in Figure 24, as the number of comparisons grows, so does the time spent on their calculation. The difference is huge, and can explain when the ND-Tree structure looses performance in some of the graphs presented before.

5.2.6 Experimental Conclusions

There are several conclusions to take from these tests. The possibility of using a database system like the one used by the NB-Tree could be an option. We can also see that the ND-Tree is much faster with the Range (Ball) Queries. This seems to be a great strength of the structure.

One of the most important conclusions to take from these tests is that if we can achieve better times with our structure coded in Java and having proved that Java is slower than C++ in this case, than we can assume that we would get even better results if we coded our structure in C++, which leads us to our future work.

6. Conclusions and Future Work

What we consider to be the major contribution of this dissertation is the dual-metric mechanism. The dimension reduction technique was the main factor for the results presented before. The structure adopted cannot be left aside since it is also important for the results, but also a consequence of the dual-metric mechanism. We chose the Quad-Tree structure because it was the best structure for our needs that were to partition a 2 dimensional space in a way that it could be used in a dynamic way. In some tests, like the range queries, we can see the advantages of having chosen the Quad-Tree structure in opposition to other structures analyzed and stated before. One aspect that wasn’t tested but is also very important is the behavior of this structure in a dynamic application. This can seem difficult because of the insertion times presented, but one must note that those times only have to happen once on the beginning of the application. After that points can be inserted one by one and that time is very low.

As stated before, in order to analyze the results in the correct way, one has to take in to consideration that our structure is coded in Java and the NB-Tree in C++ and how this affects the performance in some of the methods. Even when disregarding this difference, we can see many tests where the results are better for the ND-Tree.
Altogether, we have achieved a multidimensional indexing structure that presents better results than other current techniques, and presented a better pruning system using a dual-metric system to reduce dimensions.

The major future work would be to implement the ND-Tree in C++ to show the full potential of its structure. In this work we adopted Java because it was the primary language of the authors, and we preferred to lose time choosing the best ways to solve the problems, and not with the language itself. We think we proved that by doing this conversion we can outstand the NB-Tree in almost everything, if we can’t already.

One other detail that can be interpreted as future work is that it is necessary to define in the ND-Structure the number of points by division, and when a division reaches that number of points, it is further sub-divided. The work needed, is to define what the best value is. This value must vary according to the points in the structure, but has to be defined prior to the structure, since we don’t want to recreate all the structure every time we decide that the divisions carry too few points.

Other change that could be studied was mentioned in the conclusions of the tests, and it refers to the way that the ND-Tree reads, inserts and even keeps the points. The reading and insertion could be studied and be optimized, or even arranging a new way to do this input to the structure.

The ND-Tree also keeps the points in memory. One of the future works could be to analyze the difference between maintaining all the points in memory and using a database structure like the one used by the NB-Tree. Our structure also considers that all points have the same dimension. Since in the test files we used some examples that had different number of dimensions, we had to develop a simple solution in order to support these tests, but the solution chosen could be studied in order to achieve better solutions since no real effort was involved in the solution adopted. The solution adopted consists in: when we insert a point in the root (we refer to the root because all points are inserted in the root. It is the root that decides if the point needs to be inserted in a sub-division or not), we first have to see if that point has the dimensions equal to the max dimensions. If the points dimension is superior to the max dimension, we have to ignore that point. If the points dimension is inferior to the max dimension, we correct that point. The correction of a point consists in adding coordinates to the point. The coordinates added to the point are always zeros. We assume that if the point doesn’t have that dimension, then it is zero.

All this minor changes could bring positive results to this structure that was not contemplated since they did not figure on the objectives of this work.

7. REFERENCES

1. Yi Zuang, Yueting Zhuang, Qing Li, Lei Chen, Yi Yu, Indexing High-Dimensional Data in Dual Distance Spaces: A Symmetrical Encoding Approach, in proceedings of the International Conference on Extending Database Technology (EDBT’08)


