View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by El Servicio de Difusión de la Creación Intelectual

Optimizing the Spatial Approximation Tree from the Root

Alejandro Gómez, Verónica Ludueña, Nora Reyes
Dpto. de Informática, Universidad Nacional de San Luis, San Luis, Argentina
{*agomez, vlud, nreyes*}*@unsl.edu.ar*

Abstract

Many computational applications need to look for information in a database. Nowadays, the predominance of non-conventional databases makes the *similarity search* (i.e., searching elements of the database that are "similar" to a given query) becomes a preponderant concept.

The Spatial Approximation Tree has been shown that it compares favorably against alternative data structures for similarity searching in metric spaces of medium to high dimensionality ("difficult" spaces) or queries with low selectivity. However, for the construction process the tree root has been randomly selected and the tree ,in its shape and performance, is completely determined by this selection. Therefore, we are interested in improve mainly the searches in this data structure trying to select the tree root so to reflect some of the own characteristics of the metric space to be indexed. We regard that selecting the root in this way it allows a better adaption of the data structure to the intrinsic dimensionality of the metric space considered, so also it achieves more efficient similarity searches.

Keywords *similarity search, metric spaces, databases*

1 Introduction

The new-generation databases must handling exotic data types, such as images, fingerprints, audio clips, where the concept of exact search is of no use and we search instead for similar objects. Similarity searching has applications in a vast number of fields [19, 23]. Some examples are non-traditional databases (for example, storing images, fingerprints or audio clips, where the concept of exact search is of no use and we search instead for similar objects) [1, 22]; text searching (to find words and phrases in a text database allowing a small number of typographical or spelling errors) [20, 15]; information retrieval (to look for documents that are similar to a given query or document) [18, 2]; machine learning and classification (to classify a new element according to its closest representative) [10]; image quantization and compression (where only some vectors can be represented and we code the others as their closest representable point, as in the MPEG standard); computational biology (to find homologous regions in a DNA or protein sequence database) [21, 20]; and function prediction (to search for the most similar behavior of a function in the past so as to predict its probable future behavior).

All those applications share some common characteristics. There is a finite *dataset* of objects belonging to a *metric space*, where a *distance function* is used to assess similarity. *Similarity queries* are posed to this dataset. These consist basically in, given a new element of the space called the *query*, looking for elements of the dataset that are similar enough to the query.

Formally, let $\mathbb{U}$ be a universe of *objects*, with a non-negative *distance function* $d : \mathbb{U} \times \mathbb{U} \longrightarrow \mathbb{R}^+$ defined among them. This distance satisfies the three axioms that make $(\mathbb{U}, d)$ a *metric space*: strict positiveness ($d(x,y) = 0 \Leftrightarrow x = y$), symmetry ($d(x,y) = d(y,x)$) and triangle inequality ($d(x,z) \leq d(x,y) + d(y,z)$). The smaller the distance between two objects, the more "similar" they are. We handle a finite *dataset* $S \subseteq \mathbb{U}$, which is a subset of the universe of objects and can be preprocessed (to build an index). Later, given a new object from the universe (a *query* $q \in \mathbb{U}$), we must retrieve all similar elements in the dataset. There are two typical queries of this kind:

Range query (q, r)***:*** Retrieve all elements within distance r to q in S. That is,$\{x \in S \ , \ d(x,q) \leq r\}$.

Nearest neighbor query $k-NN(q)$***:*** Retrieve the k closest elements to q in S. That is, a set $A \subseteq S$ such that $|A| = k$ and $\forall x \in A, y \in S - A, d(x,q) \leq d(y,q)$.

The distance is assumed to be expensive to compute. Hence, it is customary to define the complexity of the search as the number of distance evaluations performed, disregarding other components such as CPU time for side computations, and even I/O time. Given a dataset of $|S| = n$ objects, queries can be trivially answered by performing n distance evaluations. The goal is to preprocess the dataset, building an *index*, such that queries can be answered with as few distance evaluations as possible. This metric space approach to handle similarity search problems is becoming widely popular [19, 23, 8].

A particular case of this problem arises when the space is a set of D-dimensional points and the distance belongs to the Minkowski L_p family: $L_p = (\sum_{1 \leq i \leq D} |x_i - y_i|^p)^{1/p}$. For example $p = 2$ yields Euclidean distance. There are effective methods to search in those spaces [12, 5] such as: *Kd–trees* [3, 4] or *R-trees* [13]. However, for roughly 20 dimensions or more those structures cease to work well. We focus in this paper on general metric spaces, although the solutions are well suited also for D-dimensional spaces. That is, the only information available are the objects and a dissimilarity function stating the distance between the objects. Moreover, regarding a D-dimensional space as a metric space reveals the true dimensionality of the dataset, which may be much lower than D, without the need of applying dimensionality reduction techniques.

The dimensionality of a vector space is the number of components of each vector. Although general metric spaces do not have an explicit dimensionality, we can talk about their *intrinsic dimensionality*, following the same idea that in vector spaces. This is a very interesting concept since the efficiency of the search methods is worse in metric spaces

with a high intrinsic dimensionality [8, 6, 7]. In these articles it is shown the analytical reason for the so called "dimensionality curse". Worthwhile to note that the concept of dimensionality is related to the "facility" or "difficulty" of searching in a D-dimensional vector space. It is said that a metric space is generally "more difficult" (intrinsic dimension higher) than another when its histogram distance is more concentrated (with larger mean as the dimension grows) than the other, making the work of any similarity search algorithm more difficult. In the extreme case we have a space where $d(x,x) = 0$ and $\forall y \neq x,\ d(x,y) = 1$, where it is impossible to avoid a single distance evaluation at search time.

Therefore, we need to count with indices to response queries efficiently. There are a number of methods to preprocess the set in order to reduce the number of distance evaluations. All those structures work on the basis of discarding elements using the triangle inequality. Note that almost all these works aim at dividing the database, inheriting from the classic divide-and-conquer ideas of searching typical data (e.g., binary search trees). A recently proposed data structure of this kind [16], which is based on a novel concept: rather than dividing the search space, approach the query spatially, that is, start at some point in the space and get closer and closer to the query. It has been shown that the *SAT* behaves better than the other existing structures on metric spaces of high dimension or queries with low selectivity, which is the case in many applications. The *SAT* is a static structure, so the construction algorithm needs to know all the elements of S in advance. Besides, it selects its root randomly. The shape of the tree and its query performance are completely determined by this choice [14, 17].

In this paper we study a possible optimization for this data structure, trying to choose a better root for the tree, in order to improve mainly the query performance. Thus, our interest was to select the tree root differently, in a way that reflects some of the characteristics of the metric space indexed. We believe that worthwhile strive to make a better choice of the tree root, and allowing that the structure adapts itself to the intrinsic dimension of metric space, so resulting in more efficient searches.

The rest of the paper is organized as follows: Section 2 introduces the Spatial Approximation Trees, Section 3 describes the methods to select the root considered, Section 4 illustrates the experimental results and their analysis, and finally Section 5 brings our conclusions and future research trends in this area.

2 Spatial Approximation Tree *(SAT)*

We describe briefly in this section the static *SAT* data structure [16]. Unlike most other structures, based on dividing the search space, the *SAT* is based on the idea of approaching the query spatially, that is, starting at some point in the space and getting closer and closer to the query. The *SAT* is experimentally shown to offer better space-time tradeoffs than other data structures in several spaces. It needs $O(n)$ space, $O(n \log^2 n / \log \log n)$ construction time, and search time: $O(n^{1-\Theta(1/\log\log n)})$ in high dimensions and $O(n^\alpha)$ $(0 < \alpha < 1)$ in low dimensions.

In order to introduce the "spatial approximation", we consider the metric space (U, d), $S \subset U$ as our database, and we concentrate on 1-NN queries (we will solve all types of queries at the end). Instead of the known algorithms to solve proximity queries by dividing the set of candidates, we try a different approach here. In our model, we are always positioned at a given element $a \in S$, randomly chosen, and try to get spatially closer to the query q (i.e., move to another element $b \in S$ which is closer to the query than the current one, that is $d(b,q) < d(a,q)$). When this is no longer possible, we are located at the nearest element to q in the set.

This approximation is performed only via "neighbors". Each element $a \in S$ has a set of neighbors $N(a)$, and we are allowed to move directly only to neighbors.

The natural structure to represent this restriction is a "directed graph" where the nodes are the elements of S and have direct edges to their neighbors. That is, there is an edge from a to b if it is possible to move from a to b in a single step. Therefore, in a vector space, the minimal graph we seek corresponds to the classical Delaunay triangulation (a graph where the elements which are Voronoi neighbors are connected). The Delaunay graph, generalized to arbitrary spaces, would be the ideal answer in terms of space complexity, and it should permit fast searching, too.

Unfortunately, it is not possible to compute the Delaunay graph of a general metric space given only the set of distances among elements of S and no further indication of the structure among elements of S. So, we make simplifications to the general idea so as to achieve a feasible solution, such that building a tree (called *SAT*) instead of a graph. Then, it is combined the spatial approximation approach with backtracking so as to answer any query $q \in U$, for both range queries and nearest neighbor queries.

The construction process of *SAT* begins with the selection of a random element $a \in S$ to be the root of the tree. Then, it is selected a suitable set of neighbors $N(a)$ satisfying:

Condition: (given $a \in S$) $\forall x \in S,\ x \in N(a) \Leftrightarrow \forall y \in N(a) - \{x\},\ d(x,y) > d(x,a)$.

That is, the neighbors of a form a set such that any neighbor is closer to a than to any other neighbor. Choosing nearest neighbors owes to the concept of getting spatially closer to the query, so that if we cannot get closer (with tolerance r) from a tree node then we can stop the search there.

We begin with the initial node a and its "bag" $B(a)$ holding all the rest of S. We first sort $B(a)$ by distance to a. Then, we start adding nodes to $N(a)$ (which is initially empty). Each time we consider a new node $b \in B(a)$, we check whether it is closer to some element of $N(a)$ than to a itself. If that is not the case, we add b to $N(a)$. At this point we have a suitable set of neighbors.

We now must decide, in which neighbor's bag we put the rest of the nodes. We put each node $b \in S - (a \cup N(a))$ in the bag $B(c)$ of its closest element c of $N(a)$. Observe that this requires a second pass once $N(a)$ is fully determined. We are done now with a, and process recursively all its neighbors, each one with the elements of its bag. So, the resulting structure is a tree that can be searched for any $q \in S$ by spatial approximation for nearest neighbor queries.

Figure 1 depicts the construction process. It is firstly invoked as `BuildTree(a,S-{a})` where a is a random element of S, selected as root. The information stored by the data structure is the root a and the $N()$ and $R()$ values of all the nodes. The covering radius $R(a)$ (i.e., the maximum distance between a and any element in the subtree

rooted by a), is used to further prune the search, by not entering subtrees such that $d(q, a) > R(a) + r$, since they cannot contain useful elements.

```
BuildTree(Node a, Set of Nodes S)
1.  N(a) ← ∅    /* neighbors of a */
2.  R(a) ← 0    /* covering radius */
3.  Sort S by distance to a (closer first)
4.  For v ∈ S
5.      R(a) ← max(R(a), d(v, a))
6.      If ∀b ∈ N(a), d(v, a) < d(v, b) Then
7.          N(a) ← N(a) ∪ {v}
8.  For b ∈ N(a)  S(b) ← ∅    /* subtrees */
9.  For v ∈ S − N(a)
10.     Let c ∈ N(a) that minimizes d(v, c)
11.     S(c) ← S(c) ∪ {v}
12. For b ∈ N(a)  BuildTree(b, S(b))
```

Figure 1: Algorithm to built a *SAT*.

Once that the tree, which allow us searching with spatial approximation, is defined we explain the range queries with radius r. The key observation is that, even if $q \notin S$, the answers to the query are elements $q' \in S$. So we use the tree to pretend that we are searching for an element $q' \in S$. We do not know q', but since $d(q, q') \leq r$, we can obtain from q some distance information regarding q' and by the triangle inequality: it is known that $\forall x \in U$, $d(x, q) \leq d(x, q') + d(q', q)$, but $d(q, q') \leq r$, so it holds that for any $x \in U$, $d(x, q) - r \leq d(x, q') \leq d(x, q) + r$. So, we are pruning all subtrees rooted at those b such that $d(q, b) > d(q, c) + 2r$.

Hence, if we search for $q \in U$ instead of simply going to the closest neighbor, we first determine the closest neighbor c of q among $\{a\} \cup N(a)$. We then enter into *all* neighbors $b \in N(a)$ such that $d(q, b) \leq d(q, c) + 2r$. This is because the virtual element q' sought can differ from q by at most r at any distance evaluation, so it could have been inserted inside any of those b nodes. In the process, we report all the nodes q' we found close enough to q.

As can be seen, what was originally conceived as a search by spatial approximation along a single path is combined now by spatial approximation along a single path is combined now with backtracking, so that we search along a number of paths.

A more sophisticated pruning criterion is obtained by noticing that all elements inserted into child c of a are not only closer to c than to a and $N(a)$, but also closer to a than to the parent of a and any neighbor of the parent of a. Extending the argument transitively, we see that c is closer to a than to any ancestor of a and to any neighbor of any ancestor of a. Let us call $A(a)$ the set of ancestors of a in the *SAT* (we include a itself in $A(a)$), and $N(A(a))$ the set of neighbors of ancestors of a (i.e., $N(A(a)) = \bigcup_{a' \in A(a)} N(a')$). Therefore, we can take c as the closest element to q among $N(A(a))$. Moreover, notice that, in an exact search for a $q \in S$, the distances between q and the nodes we traverse get reduced as we step down the tree. That is,

Observation: Let $a, b, c \in S$ such that b descends from a and c from b in the tree. Then $d(c, b) \leq d(c, a)$.

The same happens, allowing a tolerance of $2r$, in a range search with radius r. That is, for any b in the path from a to q' it holds $d(q', b) \leq d(q', a)$, so $d(q, b) \leq d(q, a) + 2r$. Hence, while at first we need to enter into all the neighbors $b \in N(a)$ such that $d(q, b) - d(q, c) \leq 2r$, when we enter into those b the tolerance is not $2r$ anymore but it gets reduced to $2r - (d(q, b) - d(q, c))$. Finally, the covering radius $R(a)$ is used to further prune the search, by not entering subtrees such that $d(q, a) > R(a) + r$, since they cannot contain useful elements.

Figures 2 and 3 illustrate more clearly the described situations, which can be found during range searches. The algorithm in Figure 4 describes the search process aforementioned.

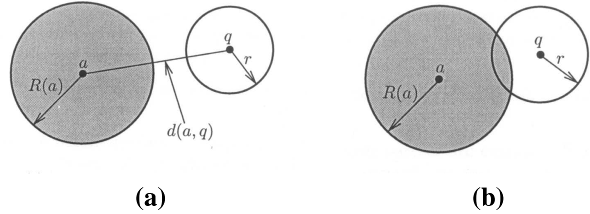

Figure 2: In **(a)** it is satisfied that $d(a, q) > R(a) + r$, so we do not have to enter in the subtree of a because any of its element will be in the range of the query. In **(b)** it is accomplished that $d(a, q) \leq R(a) + r$, so we must enter in the subtree of a because it is possible (but not sure) that any of its elements lies into the range.

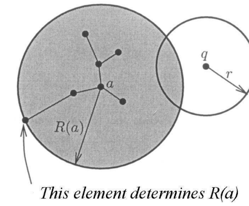

Figure 3: In this case it is fulfilled that $d(a, q) \leq R(a) + r$ and we have to enter in the subtree of a, although there is not exist any element x in this subtree such that $d(q, x) \leq r$.

```
RangeSearch(Node a, Query q, Radius r,
            Distance d_min)
1. If d(a,q) ≤ R(a) + r Then
2.     If d(a,q) ≤ r Then Report a
3.     d_min ← min {d_min} ∪ {d(q,c), c ∈ N(a)}
4.     For b ∈ N(a)
5.         If d(b,q) ≤ d_min + 2r Then
6.             RangeSearch(b, q, r, d_min)
```

Figure 4: Algorithm to search (q, r) in a *SAT* rooted at a.

In [16], it has been shown that the *SAT* gives an attractive tradeoff between memory usage, construction time, and search performance, compared against another metric data structures.

3 Improving the SAT from the Root

As it is already mentioned, the construction of the *SAT* it is completely determined by the selection of the root, that is if the same database takes another object as root, the resulting tree would be different. This selection is made randomly, then it is possible to find good roots such that the generated

tree possesses a better performance mainly during searches, but also we can find less suitable roots.

Figure 5 shows an example for a given space of two possible trees generated from the selection of different roots. However, the selection of the root at random is an attractive method because it is cheap, the choice of the root is free if we measured our costs in *number of evaluations of the distance function*. Moreover, it seems reasonable to use information from the database in order to select a good root although we pay a certain cost in distance computations, at hope that it compensates by reducing mainly the search costs, or at least the construction costs.

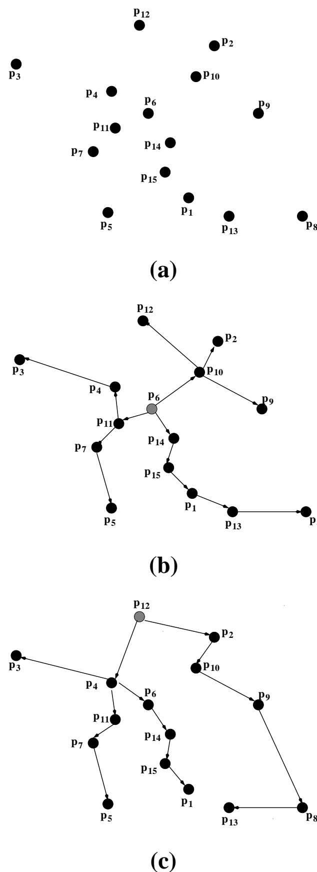

Figure 5: (a) shows a given set of objects as our space, (b) depicts the *SAT* obtained with p_6 as the tree root, (c) shows the *SAT* obtained with root p_{12}.

3.1 Selecting Roots

The assumption used in this paper is: *a object selected as root is better than another one if the tree obtained by it, possesses mainly better search costs*. Since the *SAT* is an static data structure, modifying it is extremely difficult because it does not admit insertions and deletion, we are more interested in the search costs. Moreover, the root selection is made only once before the construction of the tree.

In this paper we study different methods to handle the selection of an element as the *SAT* root, and by the experimental analysis we determine which is the method with better results. Although we are more interested in search costs, we highlight for each method the cost of selecting the root, measured in number of distance evaluations.

In some methods underlies ideas of Graph Theory, others are adaptations of methods used in other data structures to carry out similar processes and the *CSA* method (proposed in [17]) which has been designed specifically for the *SAT*.

The considered methods for root selection are: Random, *CSA* (Centroid Selection Algorithm), Sampling, M-LB-DIST (Maximum Lower Bound on DISTance), mM-LB-RAD (minimum Maximum Lower Bound RADius), and mM-AVG-RAD (minimum Maximum Average RADius).

The last four methods have been developed as adaptations of those used in the *M–tree* to select the center of a new node after an *split* operation [9]. They were adapted for being used in the selection of the roots for the *SAT*, while reasonable costs are kept. To easily identify each of those methods of we have maintained their original names.

The following is a brief description of each one. For the analysis of the costs we assume that $S \subseteq U$ is our database and $N = |S|$.

3.1.1 *CSA* Method (Centroid Selection Algorithm)

This method was proposed for selecting the root of *SAT*, it was introduced by Penarrieta, Morriberón and Cuadros-Vargas in [17]. The underlying idea is that an element which could be a good root for *SAT* would be one who is close to ideal centroid of the database. It is based on the algorithm *HF* presented at the Omni Family [11] and is reflected in the following algorithm:

1. Select a random element $s \in S$.
2. Find the farthest element e_1 from s.
3. Find the farthest element e_2 from e_1.
4. Let c the element that minimizes $|d(e_1, c) - d(e_2, c)|$ and $|d(e_1, e_2) - (d(e_1, c) + d(e_2, c))|$.

The last step is very important because there may be several elements candidates for c, but it is chosen one that minimizes the perimeter of the formed triangle between e_1, e_2, and c, and also that it is not far from the center of the triangle. Figure 6 shows the steps to develop in the algorithm.

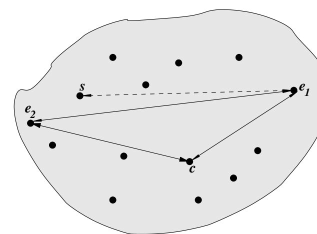

Figure 6: Geometric idea that gives rise to the *CSA* method.

3.1.2 Sampling Method

This is a random policy, but iterated over a sample of objects of size $s > 1$. The s objects are randomly selected from the database S and it is selected as the tree root the element whose maximum distance to the other $s - 1$ elements is minimum.

It is easy to see in this case that are performed $\frac{s * (s-1)}{2}$ distance evaluations, giving us a cost of $O(s^2)$. In order to keep this cost as linear regarding the cardinality

of S, we choose $s = \sqrt{N}$; and this method performs $O(N)$ distance calculations.

3.1.3 M-LB-DIST Method

This policy differs from previous ones in that it needs to use all the elements into the set. An object x is randomly selected and it is computed its distance to all of the remains elements in the set. Then, is selected as the root the farthest object from x. Formally, it is selected at random an element $x \in S$, and then is determined the object $y \in S$, such that $d(x, y) = \max_{z \in S}\{d(x, z)\}$. Clearly, this process to obtain the tree root takes about $O(N)$ distance computations.

3.1.4 mM-LB-RAD Method

It randomly chooses an object $x \in S$, and they are calculated distances between all elements $z \in S$ and x. Then, it chooses as root the object y for which $\max_{z \in S}\{|d(y, x) - d(z, x)|\}$ is minimum. This approach selects an object y that is at an intermediate distance from x.

Although the manner in which it is selected an element y as root is more complex than in the previous method, the number of distance calculations is also $O(N)$, since it only assesses the distances from all elements of S to x.

3.1.5 mM-AVG-RAD Method

This method emerged as an alternative to the previous one and tries to be neutral with respect to the information of upper and lower bounds, taking the average and applying the $\min - \max$ selection criterion. So, we choose again at random a $x \in S$, we estimate for every $z \in S$ the distance $d(x, z)$ and select an element y as root such that $\max_{z \in S}\{(d(y, x) + d(z, x))/2\}$ is minimum. This method also needs $O(N)$ distance calculations to select the root.

4 Experimental Results

To analyze the performance of each method for root selection of the *SAT*, the experiments were developed on different metric spaces, to study the behavior of each one more objectively. We consider mainly the search costs, but also we analyze what happens with construction costs, and hence the cost of the root selection. The metric spaces are:
Strings: a dictionary of 69,069 English words. The distance is the *edit distance*, that is, the minimum number of character insertions, deletions and substitutions needed to make two strings equal. This distance is useful in text retrieval spelling, typing and optical character recognition (OCR) errors.
NASA images: a set of 40,700 20-dimensional feature vectors, generated from images downloaded from NASA (`http://www.dimacs.rutgers.edu/Challenges/Sixth/software.html`).
The *Euclidean distance* is used on this space.
Color histograms: a set of 112,682 8-D color histograms (112-dimensional vectors) from an image database (`http://www.dbs.informatik.uni-muenchen.de/~seidl/DATA/histo112.112682.gz`).
Any quadratic form can be used as a distance, so we chose *Euclidean distance* as the simplest meaningful alternative.
Documents:: a set of 1,265 documents under the *Cosine similarity*, heavily used in Information Retrieval [2]. In this model the space has one coordinate per term and documents are seen as vectors in this high dimensional space. The distance we use is the angle (arccos of inner product) among the vectors. The documents are the files of the TREC-3 collection (`http://trec.nist.gov`).
Vectors: a space of 100,000 vectors in the real unitary cube in dimension 15, using *Euclidean distance*. We generated 100,000 random points with uniform distribution $[0, 1]^{15}$. This is a hard space (concentrate histogram of distances). We treat this just as a metric space, disregarding coordinate information. This choice allows us to control the exact dimensionality we are working with, which is not so easy if is a general or real metric space.

The experiments consist in building a *SAT* for each method of root selection on every space considered, and then we search on they. In all cases, we built the indices with 90% of the objects of metric space and used the other 10% (randomly chosen) as queries. All our results are the average of 4 executions of each experiment using different dataset permutations.

Strings Space is the only one with a discreet distance function, so we used radii 1 to 4, which retrieved on average $0,00003\%$, el $0,00037\%$, el $0,00326\%$ y el $0,01757\%$ respectively. For the other spaces with continuous distance functions we have considered range queries retrieving on average $0,01\%$, $0,1\%$, 1% of the dataset. This corresponds to the following radii:**a)** NASA Images Space: 0.605740, 0.78 y 1.009,**b)** Color Histograms Space: 0.051768, 0.082514 y 0.131163,**c)** Documents Space: 0.189441, 0.222466 y 0.600048, and **d)** Vectors Space: 0.686576, 0.833130 y 1.019767.

4.1 Comparing the Different methods

We compare every method in each particular space to observe which of them performs fewer of distance evaluations during searches. They was added in the plots the results of methods of minimum and maximum distances that although were not considered as alternative methods in this study because they are too expensive, they serve as reference for other methods.

4.1.1 Strings Space

The comparison shows that, of all methods considered, those with lower search costs are *CSA* and Sampling. Noted that the Random selection method of the root does not seem to get a good tree, from the standpoint of searches. Since, almost all other ways of selecting the root, overcome it for the four radii considered. Figure 7 shows these results.

4.1.2 NASA Images Space

In this space the method that had better performance in searches is the M-LB-DIST. The methods mM-LB-RAD and mM-AVG-RAD obtained identical performances. Again, here it can see that the Random method does not produce the best tree. Figure 8 illustrates this analysis.

4.1.3 Color Histograms Space

In this metric space we found that the most cost-effective method for searches was the Random method and then, with very little difference, the *CSA* method. It is striking to note

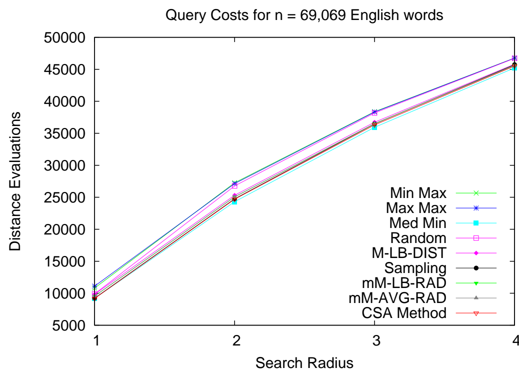

Figure 7: Comparison of search costs on Strings Space.

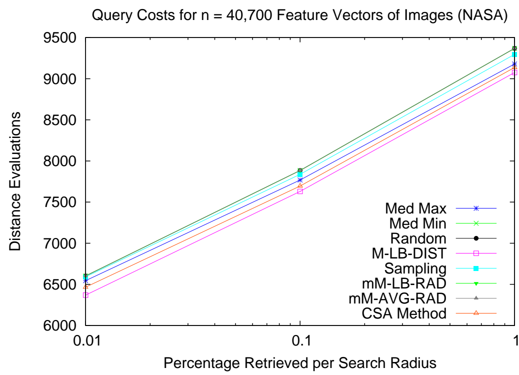

Figure 8: Comparison of search costs on NASA Images.

that the Random method produces the lowest search costs in distance evaluations, though it is an algorithm that makes $O(1)$ distance calculations and extremely simple. Undoubtedly the characteristics of this space make that it was unproductive the use of our techniques, but clearly this has not been the case in most of other spaces used. Figure 9 reflects these facts.

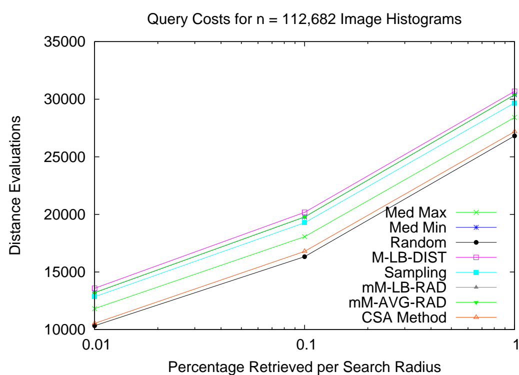

Figure 9: Comparison of search costs on Color Histograms.

4.1.4 Documents Space

In this space, the best method in searches for all radii and also obtaining a good construction costs is the *CSA* method, although the differences between *CSA* and the other methods are not too significant. We can observe that these differences between the best and the worst cost, for all search radii, is close to 50 distance evaluations. In Figure 10 it is possible to note this situation.

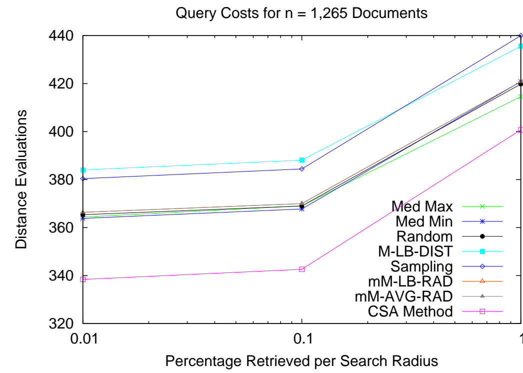

Figure 10: Comparison of search costs on Documents.

4.1.5 Vectors Space

In the experiments we can observe that there is a great similarity between the search costs for almost all methods, actually there is not differentiation between them. The only method that stands out as the fewest costly, with small differences in the number of distance evaluations in respect to other methods, is *CSA* in all radii considered. However, the *CSA* method is the worst in terms of construction costs. If you are not interested in paying so much in construction, Sampling would be a good option because their performance is so similar in searches and its construction cost is much lower. Figure 11 depicts these results graphically.

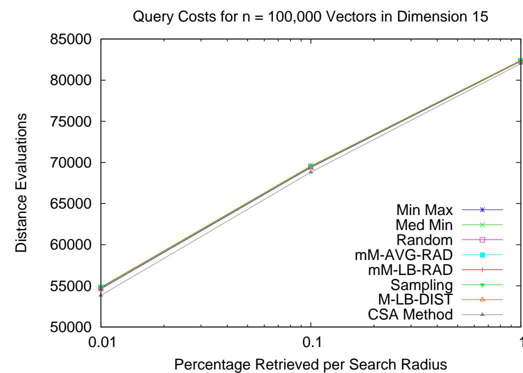

Figure 11: Comparison of search costs on Vectors.

5 Conclusions and Future Work

The Spatial Approximation Tree has been shown that it compares favorably against alternative data structures for similarity searching in metric spaces of medium to high dimensionality ("difficult" spaces) or queries with low selectivity[16]. A detailed analysis shows that the selection of the tree root is one of the processes that could be optimized, because the *SAT* is completely determined by it and this selection originally was done at random [14]. Other authors have been devoted to this topic [17]. However we believe that our work provides more information because we

studied and compared more methods, most of which were designed or adapted by us for this structure, and more experiments were conducted.

We have achieved to build a tree more efficient for searches, only paying some additional distance calculations regarding the original *SAT* and maintaining the correctness of the data structure. We experimentally demonstrate that greater knowledge about the particular metric space allows us improving the data structure. The algorithms proposed for selecting the root permit *to improve* both search and construction costs with respect to the original *SAT*.

While our results are applicable mainly to *SAT*, some of them might be adequate to other arboreal data structures. In the same way that we have improved the static version of *SAT* selecting the tree root in a different form to the original one, we could try to adapt some of the methods to the dynamic version of *SAT*. So far the *SAT* works in main memory; so the possibility that the selection of a better root gives us as a result a more balanced tree, could allow to store it efficiently in the secondary memory.

References

[1] P. Apers, H. Blanken, and M. Houtsma. *Multimedia Databases in Perspective*. Springer, 1997.

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

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

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

[5] C. Böhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, Sept. 2001.

[6] E. Chávez and G. Navarro. Measuring the dimensionality of general metric spaces. Tech. Report TR/DCC-00-1, DCC, University of Chile, 2000.

[7] E. Chávez and G. Navarro. Towards measuring the searching complexity of metric spaces. In *Proc. Mexican Computing Meeting*, II, 969–978, México, 2001.

[8] E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, Sept. 2001.

[9] P. Ciaccia, M. Patella, and P. Zezula. M-tree: an efficient access method for similarity search in metric spaces. In *Proc. of the 23rd Conference on Very Large Databases*, 426–435, 1997.

[10] R. Duda and P. Hart. *Pattern Classification and Scene Analysis*. Wiley, 1973.

[11] R.F. Santos Filho, A. Traina, C. Traina Jr., and C. Faloutsos. Similarity search without tears: The omni family of all-purpose access methods. In *Proc. of the 17th International Conference on Data Engineering*, 623–630, 2001.

[12] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

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

[14] G. Hjaltason and H. Samet. Improved search heuristics for the sa-tree. *Pattern Recognition Letters*, 24(15):2785–2795, 2003.

[15] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, 1992.

[16] G. Navarro. Searching in metric spaces by spatial approximation. *The Very Large Databases Journal*, 11(1):28–46, 2002.

[17] J. Peñarrieta, P. Morriberón, and E. Cuadros-Vargas. Distributed spatial approximation tree - sat∗. In M. Marin and G. Acuña, editors, *Actas de la XXXII Conferencia Latinoamericana de Informatica (CD ROM)*, August 2006. 956-303-028-1.

[18] G. Salton and M. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[19] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc.,USA, 2005.

[20] D. Sankoff and J. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*. Addison-Wesley, 1983.

[21] M. Waterman. *Introduction to Computational Biology*. Chapman and Hall, 1995.

[22] A. Yoshitaka and T. Ichikawa. A survey on content-based retrieval for multimedia databases. *IEEE Trans. on Knowledge and Data Engineering*, 11(1):81–93, 1999.

[23] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach*, volume 32 of *Advances in Database Systems*. Springer, 2006.