Slim-trees: High Performance Metric Trees Minimizing Overlap Between Nodes

Caetano Traina Jr.[1], Agma Traina[2], Bernhard Seeger[3], Christos Faloutsos[4]

October, 1999
CMU-CS-99-170

School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3890

[1] Department of Computer Science, University of São Paulo at São Carlos - Brazil
Caetano@cs.cmu.edu. His research was partially funded by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo - Brazil, under Grant 98/05556-5). On leave at Carnegie Mellon University.

[2] Department of Computer Science, University of São Paulo at São Carlos - Brazil .
Agma@cs.cmu.edu. Her research was partially funded by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo - Brazil, under Grant 98/0559-7). On leave at Carnegie Mellon University.

[3] Fachbereich Mathematik und Informatik, Universität Marburg - Germany.
Seeger@mathematik.uni-marburg.de. His work has been supported by Grant No. SE 553/2-1 from DFG (Deutsche Forschungsgemeinschaft).

[4] Department of Computer Science, Carnegie Mellon University - USA.
Christos@cs.cmu.edu. This material is based upon work supported by the National Science Foundation under Grants No. IRI-9625428, DMS-9873442, IIS-9817496, and IIS-9910606, and by the Defense Advanced Research Projects Agency under Contract No. N66001-97-C-8517. Additional funding was provided by donations from NEC and Intel. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, DARPA, or other funding parties.

Keywords: metric databases, metric access methods, index structures, multimedia databases.

Abstract

In this paper we present the Slim-tree, a dynamic tree for organizing metric datasets in pages of fixed size. The Slim-tree uses the "fat-factor" which provides a simple way to quantify the degree of overlap between the nodes in a metric tree. It is well-known that the degree of overlap directly affects the query performance of index structures. There are many suggestions to reduce overlap in multi-dimensional index structures, but the Slim-tree is the first metric structure explicitly designed to reduce the degree of overlap.

Moreover, we present new algorithms for inserting objects and splitting nodes. The new insertion algorithm leads to a tree with high storage utilization and improved query performance, whereas the new split algorithm runs considerably faster than previous ones, generally without sacrificing search performance. Results obtained from experiments with real-world data sets show that the new algorithms of the Slim-tree consistently lead to performance improvements. For range queries, we observed improvements up to a factor of 35%.

1. Introduction

With the increasing availability of multimedia data in various forms, advanced query processing techniques are required in future database management systems (DBMS) to cope with large and complex databases. Of utmost importance is the design of new access methods which support queries like similarity queries in a multimedia database. While there has been a large number of proposals for multidimensional access methods [GG 98], almost all of them are not applicable to multimedia databases since they assume that data belong to a multidimensional vector space. However, data of multimedia databases often are not in vector spaces, but in metric spaces.

In this paper we address the problem of designing efficient metric access methods (MAM). An MAM should organize a large set of objects in a dynamic environment assuming only the availability of a distance function d which satisfies the three rules of a metric space (symmetry, non-negativity and triangle inequality). Consequently, an MAM is not permitted to employ primitive operations like addition, subtraction or any type of geometric operation. While insertions of new records should be supported efficiently, we are mainly interested in MAMs supporting range queries and similarity queries (nearest neighbor queries). Efficiency of an MAM is determined by several factors. First, since the data set is generally too large to be kept in main memory, one major factor for efficiency is the number of disk accesses required for processing queries and insertions. We assume here that an MAM organizes data in pages of fixed size on disk and that disk access refers to the time to read (write) one page from disk into main memory. Second, the computational cost of the distance function can be very high such that the number of distance calculations has a major impact on efficiency. We expect, however, that there is a strong relationship between the number of disk accesses and the number of distance calculations. Third, storage utilization is another important factor although it has rarely been considered in this context previously. The reason we are concerned about storage utilization is not because of the storage cost, but primarily because of the number of disk accesses required to answer "large" range queries. For those queries, the number of accesses is only low when the storage utilization is sufficiently high. In other words, an MAM can perform less efficiently than a simple sequential scan for such cases.

the Slim-tree, a new dynamic MAM In this paper we present. The Slim-tree shares the basic data structure with other metric trees like the M-tree [CPZ 97] where data is stored in the leaves and an appropriate cluster hierarchy is built on top. The Slim-tree differs from previous MAMs in the following ways. First, a new split algorithm based on the Minimum Spanning Tree (MST) is presented which performs faster than other split algorithms without sacrificing search performance of the MAM. Second, a new algorithm is presented to guide an insertion of an object at an internal node to an appropriate subtree. In particular, our new algorithm leads to considerably higher storage utilization. Third, and probably most important, the Slim-down algorithm is presented to make the metric tree tighter and faster in a post-processing step. This algorithm was derived from our 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 objects (e.g. circles in a two-dimensional space) cannot be used for metric data. Instead, we present the "fat-factor" and the "bloat-factor" to measure the degree of overlap where a value close to zero indicates low overlap. We show that the Slim-down algorithm reduces the bloat-factor and hence, improves the query performance of the metric tree.

The remainder of the paper is structured as follows. In the next section, we first give a brief history of MAMs, including a concise description of the datasets we used in our experiments. Section 3 introduces the Slim-tree, and Section 4 presents its new splitting algorithm based on minimal spanning trees. Section 4 introduces the fat-factor and the bloat-factor. The Slim-down algorithm is described in Section 6, while Section 7 gives a performance evaluation of the Slim-tree. Section 8 presents the conclusion of this paper.

2. Survey

The design of efficient access methods has interested researchers for more than three decades. An excellent survey on multidimensional access methods is given in [GG 98]. However, most of these access methods require that data be ordered in a one- or multi-dimensional vector space.

The problem of supporting nearest neighbor and range queries in metric spaces has recently attracted the attention of researchers. The pioneering work of Burkhard and Keller [BK73] provided different interesting techniques for partitioning a metric data set in a recursive fashion where the recursive process is materialized as a tree. The first technique partitions a data set by choosing a representative from the set and grouping the elements with respect to their distance from the representative. Originally Burkhard and Keller in [BK 73] assumed that the distance function returns integral values, but the approach can also be extended to numerical values. The second technique partitions the original set into a fixed number of subsets and chooses a representative from each of the subsets. The representative and the maximum distance from the representative to a point of the corresponding subset are also maintained, to support nearest neighbor queries. The metric tree of Uhlmann [Uhl 91] and the vantage-point tree (vp-tree) of Yanilos [Yia 93] are somehow similar to the first technique of [BK 73] as they partition the elements into two groups according to a representative, called a vantage point. In [Yia 93] the vp-tree has also been generalized to a multi-way tree. In order to reduce the number of distance calculations, Baeza-Yates et al [BCM 94] suggested to use the same vantage point in all nodes that belong to the same level. Then, a binary tree degenerates into a simple list of vantage points. Another method of Uhlmann [Uhl 91] is the generalized hyper-plane tree (gh-tree). The gh-tree partitions the data set into two by picking two points as representatives and assigning the remaining to the closest representative. Bozkaya and Ozsoyoglu [BO 97] proposed an extension of the vp-tree called multi-vantage-point tree (mvp-tree) which chooses in a clever way m vantage points for a node which has a fanout of m^2. The Geometric Near Access Tree (GNAT) of Brin [Bri 95] can be viewed as a refinement of the second technique presented in [BK 73]. In addition to the representative and the maximum distance, it is suggested that the distances between pairs of representatives be stored. These distances can be used to prune the search space using the triangle inequality.

All methods presented above are static, in the sense that the data structure is built once and new insertions are not supported. The M-tree [CPZ 97] of Ciaccia, Patella and Zezulla overcomes this deficiency. The M-tree is a height-balanced tree where the data elements are stored in the leaves. An internal node consists of a set of entries where each consists of a pointer to a subtree, a representative and a distance d which is an upper limit for the maximum distance between the representative and an element in the leaves of the corresponding subtree. Each of the nodes is filled with at least c and at most C entries, where $c \leq C/2$. The M-tree supports insertions similar to R-trees [Gut 84].

In order to illustrate the performance of the different MAM, we use six synthetic and real datasets throughout

Datasets	Number of Objects N	Dimension D	Metric	Description
Uniform2D	10,000	2	L_2	Uniformly distributed data (see Figure1a)
Sierpinsky	9,841	2	L_2	Fractal dataset (see Figure 1b)
MGCounty	15,559	2	L_2	Intersection points of roads from Montgomery County - Maryland (see Figure 1c).
EigenFaces	11,900	16	L_2	Face vectors from the Informedia project [WTS 96].
FaceIT	1,056	unknown	FaceIt ™	A dataset constructed by a distance matrix obtained from FaceIt™ software, version 2.51 [Vis 98].

Table 1 - Datasets used in the experiments.

the paper. Table 1 reports the most important characteristics of our datasets. Among them there are vector datasets (2- and 16-dimensional) and metric datasets. Specifically note that for the FaceIT dataset we have used a distance function from a commercial software product. In general, we have used the L_2 metric for the vector datasets, but for the EnglishWords dataset, Levenshtein or string edit distance (L_{Edit}) was used. L_{Edit} (x,y), is a metric which counts the minimal number of symbols that have to be inserted, deleted, or substituted, to transform x into y (e.g. L_{edit} (“head”, “hobby”) = 4 - three substitutions and one insertion).

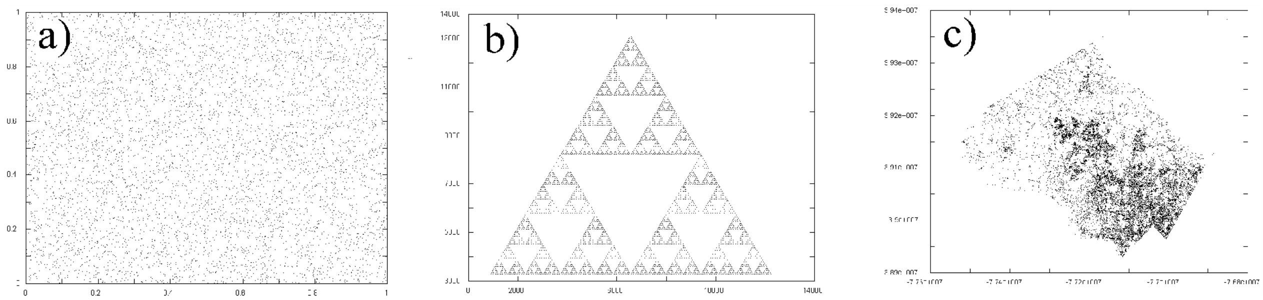


Figure 1. The two-dimensional vector datasets used in the experiments. a) Uniform2D. b) Sierpinsky triangle. c) MGCounty.

Symbols	**Definitions**
$d(x,y)$	the distance function between objects x and y
T	a metric tree
N	number of objects in the dataset
M	number of nodes in a metric tree
M_{min}	minimal number of nodes for a given metric tree
H	height of the metric tree
H_{min}	minimal height for a given metric tree
T	metric tree
I_C	total number of node accesses required to answer a pont query for each object
C	Effective capacity of a metric tree node (average number of points stored in a non-root node)
$fat(T)$	fat-factor for the metric tree T
$bl(T)$	bloat-factor for the metric tree T

Table 2 - Summary of symbols and definitions.

3. The Slim-Tree: an improved performance metric tree

The Slim-tree is a balanced and dynamic tree that grows bottom-up from the leaves to the root. Like other metric trees, the objects of the dataset are grouped into fixed size disk pages, each page corresponding to a tree node. The objects are stored in the leaves. The main intent is to organize the objects in a hierarchical structure using a representative as the center of each minimum bounding region which covers the objects in a sub-tree.

The Slim-tree has two kinds of nodes, data nodes (or leaves) and index nodes. As the size of a page is fixed, each type of node holds a predefined maximum number of objects C. For simplicity, we assume that the capacity C of the leaves is equal to the capacity of the index nodes. Table 2 summarizes the symbols used in this paper.

The leaf nodes hold all objects stored by the Slim-tree, and their structure is

leafnode [array of $<Oid_i, d(O_i, Rep(O_i)), O_i>$]

where, Oid_i is the identifier of the object O_i and $d(O_i, P(O_i))$ is the distance between the object O_i and the representative object of this leaf node $Rep(O_i)$.

indexnode [array of $<O_i, Radius_i, d(O_i, Rep(O_i)), Ptr(TO_i), NEntries(Ptr(TO_i))>$]

where, O_i keeps the object that is the representative of the sub-tree pointed by $Ptr(TO_i)$, and $Radius_i$ is the covering radius of that region. The distance between O_i and the representative of this node $Rep(O_i)$ is kept in $d(O_i, Rep(Oi))$. The pointer $Ptr(TO_i)$ points to the root node of the subtree rooted by O_i. The number of entries in the node pointed by $Ptr(TO$i$)$ is held by $NEntries(Ptr(TO_i))$.

Analogous to other metric trees, the distance to a representative can be used in combination with the triangle inequality to prune an entry without any extra distance calculation. The regions that corresponds to each node of the Slim-tree can overlap each other. The increasing of overlaps also enlarges the number of paths to be traversed when a query is issued, so it also increases the number of distance calculations to answer queries. The Slim-tree was developed to reduce the overlapping between regions in each level.

3.1 - Building the Slim-tree

The objects are inserted in a Slim-tree in the following way. Starting from the root node, the algorithm tries to locate a node that can cover the new object. If none qualifies, select the node whose center is nearest to the new object. If more than one node qualifies, execute the *ChooseSubtree* algorithm to select one of them. This process is recursively applied for all levels of the tree. When a node *m* overflows, a new node *m'* is allocated at the same level and the objects are distributed among the nodes. When the root node splits, a new root is allocated and the tree grows one level.

The Slim-tree has three options for the *ChooseSubtree* algorithm: <u>random</u> (randomly choose one of the qualifying nodes), <u>mindist</u> (choose the node that has the minimum distance from the new object and the center of the node), <u>minoccup</u> (choose the node that has the minimum occupancy among the qualifying ones). The number of entries in each child node (*NEntries*) maintained in each *indexnode* of the Slim-tree is intended to be used by the *MinOccup ChooseSubtree* algorithm. Although it uses some memory space in each *indexnode*, this is usually a small proportion of the total memory used for each entry (one byte is usually enough), but as we will show later, this *ChooseSubtree* algorithm generated trees with higher node occupation rates, leading to smaller number of disk accesses.

The splitting algorithms for the Slim-tree are:

Random - The two new center objects are randomly selected, and the existing objects are distributed among them. Each object is stored in a new node that has its center nearest this object, with respect to a minimum utilization of each node. This is not a wise strategy, but it is very fast.

minMax - All possible pairs of objects are considered as potential representatives. For each pair, a linear algorithm assigns the objects to one of the representatives. The pair which minimizes the covering radius is chosen. The complexity of the algorithm is $\Theta(C^3)$, using $\Theta(C^2)$ distance calculations. This algorithm has already been used for the M-tree, and it was found to be the most promising splitting algorithm regarding query performance [CP 98].

MST - The minimal spanning tree [Kru 56] of the objects is generated, and one of the longest arcs of the tree is dropped.This algorithm is one of the contributions of this paper, and it is presented next. This algorithm produces Slim-trees almost as good as the minMax algorithm, in a fraction of the time.

4. The splitting algorithm based on minimal spanning tree

In this section we address the following problem. Given a set of *C* objects in a node to be split, quickly divide them in two groups, so that the resulting Slim-tree leads to low search times. We propose an algorithm based on the minimum spanning tree [Kru 56], which has been successful in clustering. We consider the full graph consisting of *C* objects and *C(C-1)* edges, where the weight of the edges refers to the distance between the connecting objects. So, we proceed with to the steps as showed in Figure 2.

```
Algorithm 1 - Splitting of a node using MST
begin
        1. Build the MST on the C objects of the node.
        2. Delete the longest edge.
        3. Report the connected components as two groups.
        4. Choose the representative of each group, i.e., the object whose maximum distance to all other objects
                of the group is the shortest.
end
```

Figure 2. Algorithm for splitting a node using minimal spanning tree.

Unfortunately, this algorithm does not guarantee that each group will receive a minimum percentage of objects. To obtain more even distribution, we choose from among the longest arcs the most appropriate one. If none exists (as in a star-shaped set), we accept the uneven split and remove the largest edge. The execution time of this algorithm is O(C^2.log (C)) on the number of node objects.

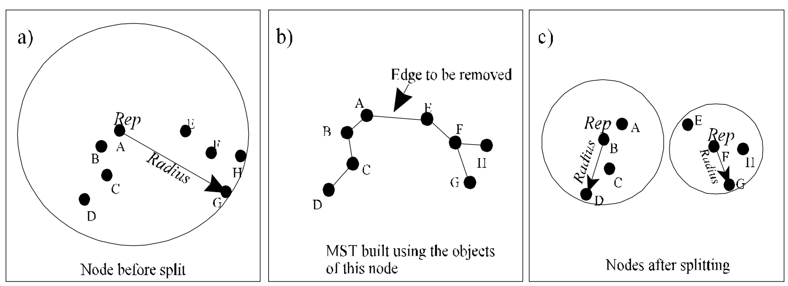


Figure 3. Exemplifying a node split using the MST algorithm.

Figure 3 illustrates our approach applied to a vector space. After building the MST, the edge between objects *A* and *E* will be deleted, and one node will keep the objects *A, B, C, D*, having *B* as the representative. The other node will have the objects *E, F, G* and *H*, having *F* as the representative.

5. Overlap optimization

In this section we present the theoretical underpinnings behind the Slim-down algorithm. The Slim-down algorithm is an easy-to-use approach to reduce overlaps in an existing Slim-tree. Before presenting the algorithm, we have to define the meaning of overlap in a metric space. It should be noted that the notion of overlap in vector space cannot be applied to a metric space [CPZ 98].

A typical assumption when someone is estimating the number of distance calculations or disk accesses from a tree, is that the tree is 'good' [FK 94][TTF 99]. That is, the nodes are tight and the overlaps over the nodes are minimal. The present work directly tackles this subject. That is, the major motivation behind this work was to solve the following problem: *"Given N objects organized in a metric tree, how can we express its 'goodness'/'fitness', with a single number ?"*

We also show that our approach to measuring overlap in a metric space leads to the fat-factor and to the bloat-factor. Both of these factors are suitable to measure the goodness of the Slim-tree and other metric trees. After discussing the properties of these factors, we will present the Slim-down algorithm.

5.1. Computing Overlap in a Metric Access Method

Let us consider two index entries stored in a node of the Slim-tree. In vector spaces, the overlap of two entries refers to the amount of the common space which is covered by both of the bounding regions. When the data is in a vector space, we simply compute the overlap as the volume of the intersection. Since the notion of volume is not available in a metric space we pursue a different approach. Instead of measuring the amount of space, we suggest counting the number of objects in the corresponding sub-trees which are covered by both regions.

Definition 1 - Let I1 and I2 be two index entries. The overlap of I1 and I2 is defined as the number of objects in the corresponding sub-trees which are covered by both regions divided by the num of the objects in both sub-trees.

This definition provides a generic way to measure the intersection between regions of a metric tree, enabling the use of the optimization techniques, developed for vector spaces, on metric trees.

5.2. The Fat-Factor

Analogous to Definition 1 of overlap in a metric space, in this section we present a method to measure the goodness of a metric tree. The basic idea of the following definition of the fat-factor is that a good tree has very little or ideally no overlap between its index entries. Such an approach is compatible with the design goals of index structures like the R+-tree [SRF 87] and the R*-tree [BKS+ 90], which were designed with the goal of overlap minimization.

Our definition of the fat-factor makes two reasonable assumptions. First, we take into account only range queries to estimate the goodness of a tree. This assumption is not restrictive since nearest neighbor queries can be viewed as special cases of range queries [BKK+ 97]. Second, we assume that the distribution of the centers of range queries follows the distribution of data objects. This seems to be reasonable since we expect the queries are more likely issued in regions of space where the density of objects is high.

Assuming the above, it is easy to state how an ideal metric-tree should behave. For a point query (a range query with radius zero), the ideal metric-tree requires that one node be retrieved from each level. Thus, the fat-factor should be zero. The worst possible tree is the one which requires the retrieval of all nodes to answer a point query. In this situation the fat-factor should be one. From this discussion we suggest the following definition of fat-factor.

Definition 2 - Let T be a metric tree with height H and M nodes, $M \geq 1$. Let N be the number of objects. Then, the *fat-factor* of a metric tree T is

where I_C denotes the total number of node accesses required to answer a point query for each of the N objects

$$fat(T) = \frac{I_C - H * N}{N} \cdot \frac{1}{(M - H)} \qquad \textbf{(1)}$$

stored in the metric tree.

Lemma 1 - Let T be a metric tree. Then, *fat(T)* returns a value in the range [0,1]. The worst possible tree returns one, whereas an ideal tree returns zero.

Proof: Let us consider a point query for an object stored in the tree. Such a query has to retrieve at least one node from each level of the tree. In particular, the nodes on the insertion path of the object qualify and are required to be read from disk into memory. A lower limit for I_C (the total number of disk accesses for all point queries) is then $H*N$ resulting in a fat factor of zero. The worst case occurs when each node has to be read for each of the queries. An upper limit of I_C is then $M*N$ resulting in a fat-factor of one. Since the fat factor is a linear function in I_C and $H*N \le IC \le M*N$, it follows that the fat factor has to be in the range [0,1].

QED

In order to compute the fat-factor of a metric tree, we need to process a point query for each of the objects stored in the metric tree. The computation cost depends on the fat-factor since we need only to investigate those sub-trees where the corresponding index entry qualifies. Therefore, this operation requires at most $N*M$ comparisons and distance computations when the tree is the worst possible, and only O($N*log\ N$) for an ideal tree. The number of disk accesses depends on the available buffer space, but in the worst case we need to retrieve every page once for each point query. In order to reduce the complexity of the computation of the fat-factor, sampling might be considered as well. Therefore, complexity does not depend on N, but on the size of the sample.

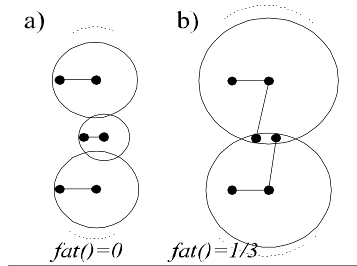


Figure 4. Two trees storing the same dataset with different number of nodes and fat-factors. Root nodes are shown in broken line.

Figure 4 shows two trees and their fat-factor. In order to illustrate the relationships between the representative and its associated objects, we have drawn a connection line. Calculating the fat-factor for these trees is straightforward, e.g., for the tree in Figure 4a we have I_C =12, H=2, N=6 and M=4, leading to a fat-factor=0. For the tree in 4b we have I_C =14, H=2, N=6 and M=3, leading to a fat-factor=1/3.

5.3. Comparing different trees for the same dataset: the bloat-factor

The fat-factor is a measure of the amount of objects that lie inside intersection of regions defined by nodes at the same level of a metric tree. If two trees store the same dataset and have the same number of nodes but have different fat-factors, the tree with the smaller factor will have fewer points in intersecting regions, and thus it will need fewer disk accesses and distance calculations to perform a given query. However, if two trees storing the same dataset have different number of nodes, the direct comparison of the corresponding fat-factors will not give such an indication. This is due to the fact that a tree with fewer nodes can lead to a tree with more objects lying inside intersection regions, and thus a bigger fat-factor. However the average number of disk accesses needed to answer the queries can also be smaller because there are less nodes to be read (see Figure 4).

To enable the comparison of two trees that store the same dataset (but that use different splitting and/or different promotion algorithms leading to different trees), we need to "penalize" trees that use more than the minimum required number of nodes (and so disk pages). This can be done by defining a new measure, called the "bloat-factor". In a similar way to the fat-factor, the bloat-factor considers not the height and number of nodes in the real tree, but that of the minimum tree. Among all possible trees, the minimum tree is the one with minimum possible height H_{min} and minimum number of nodes $M_{\min}$. Thus, this leads to the following definition.

Definition 3 - The *bloat-factor* of a metric tree T with more than one node is expressed as

$$bl(T) = \frac{I_C - H_{\min} * N}{N} \cdot \frac{1}{(M_{\min} - H_{\min})} \tag{2}$$

This factor will vary from zero to a positive number that can be greater than one. Although not limited to one, this factor enables the direct comparison of two trees with different bloat-factors, as the tree with the smaller factor always will lead to a lesser number of disk accesses.

The minimum height of the tree organizing N objects is $H_{\min} = \lceil \log_C N \rceil$ and the minimum number of nodes for a given dataset can be calculated as $M_{\min} = \sum_{i=1}^{H_{\min}} \lceil N / C^i \rceil$ where, C is the capacity of the nodes.

It is worth emphasizing that both the fat-factor and the bloat-factor are directly related to the average amount of overlap between regions in the same level of the tree, represented by I_c. The fat-factor measures how good a given tree is with respect to this amount of overlaps, regardless of a possible waste of disk space due to a lower occupation of its nodes. The bloat-factor enables us to compare two trees, considering both the amount of overlaps and the efficient occupation of the nodes.

6. The Slim-down algorithm

In this section we present an algorithm that produces a 'tighter' tree. The fat and bloat-factor indicate whether a tree has room for improvement. It is clear from Definition 3 that if we want to construct a tree with a smaller bloat-factor, we need first to diminish the number of objects that fall within the intersection of two regions in the same level. Secondly, we may need to decrease the number of nodes in the tree.

We propose a Slim-down algorithm to post-process a tree, aiming to reduce these two numbers in an already constructed tree. This algorithm is described in Figure 5 and Figure 6 illustrates graphically it.

Algorithm 2 - Slim down
begin

1. For each node i in a given level of the tree, find the farthest object c from the representative b.
2. Find a sibling node j of i, that also covers object c. If such a node j exists and it is not full, remove c from node i and insert it into node j. Correct the radius of node i.
3. Steps 1 and 2 must be applied sequentially over all nodes of a given level of the tree. If after a full round of these two steps, an object moves from one node to another, another full round of step 1 and 2 must be re-applied.

end

Figure 5. Slim down algorithm.

In this way, if object c was moved from node i to node j in step 2, and it is the only object in node i at this distance from the original center, then the correction of the radius of node i will reduce the radius of this node without increasing any other radii. As Figure 6 illustrates, we can assume that object e is the next farthest object from the representative of node i. Thus, after the reduction, the new radius of node i will be that shown with broken line. With this reduction, at least object c will go out of the region of this node which intersects with the region of node j, reducing I_C counting. Moreover, when this algorithm is applied, we do not guarantee a minimum occupancy in the nodes of the tree, so eventually some nodes can become empty, further reducing the number of nodes in the tree. It must be noted that step 2 can take advantage of the triangle inequality to prune part of the needed distance calculations.

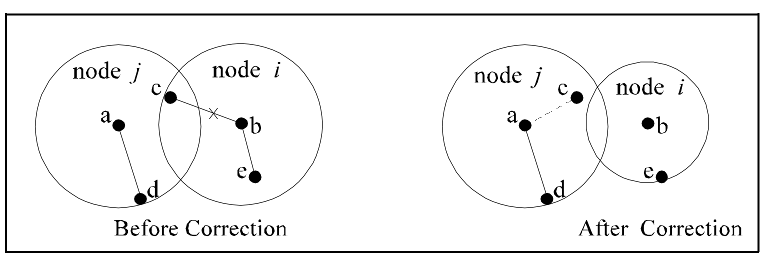


Figure 6. How the Slim-down algorithm works.

A difficulty can happen in step 3 if the situation shown in Figure 7 occurs. In this case, objects *f*, *d* and *e* will synchronously move from nodes *i*, *j* and *k* to nodes *j*, *k*, and *i* respectively, and then again to their original positions. This is illustrated in Figure 7 by the sets of solid and broken lines. As this can lead to an infinite loop, we limited the number of executions of step 3 to three times the number of objects in the node in the preceding level, which holds the moving objects. The experiments performed indicated this value as a good choice.

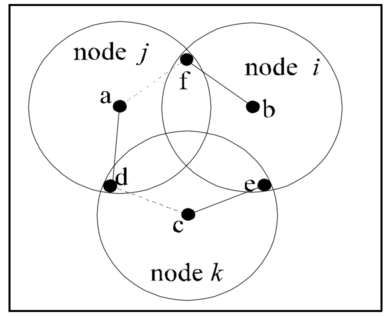


Figure 7. A cyclic move of objects without reducing radii.

We implemented the algorithm to manipulate the leaf nodes after indexing the full dataset. Figure 8 plots the regions of the leaves of trees created with the Sierpinsky dataset, using the random splitting algorithm. Figure 8a shows the tree prior to the slimming down, and Figure 8b shows it after correction by the Slim-down algorithm. The regions shown correspond to the minimum bounding circles at the leaf level only (to avoid cluttering).

The tree of Figure 8b clearly has fewer and tighter circles; therefore, it should perform better. This is confirmed by the bloat-factor values, which are 0.03 for the tree in Figure 8a, and 0.01 for the tree in Figure 8b. We used the Sierpinsky triangle dataset in this example due to its well-known shape, so that seeing the improvements would be easier. However, higher or metric (non-dimensional) datasets usually lead to more effective improvements.

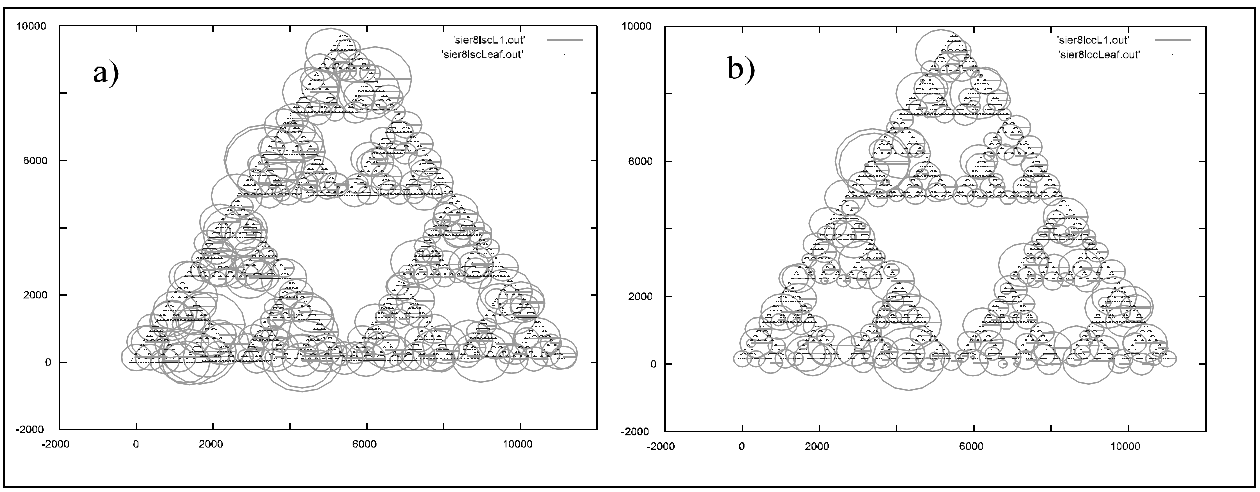


Figure 8. A tree indexing the Sierpinsky triangle using the random splitting algorithm: a) before the correction the bloat-factor is 0.03, and b) after the correction the bloat-factor is 0.01.

This algorithm can be executed at different phases of the evolution of the tree. The following variations immediately come to mind.

a) A similar algorithm could be applied to the higher levels of the tree.

b) The algorithm could be dynamically applied to slim down the sub-tree stored in a node just after one of its direct descendants has been split.

c) When a new object must be inserted in a node which is full, a single relocation of the farthest object of one node could be tried instead of splitting.

Besides the algorithm being applied over the leaf nodes after the completion of the tree, we also have implemented variation (b), and we found that it indeed leads to a better tree. Moreover, both variations can be applied isolated or together, and either way, each provides an increase in performance. However, this last variation slows down the building of the tree and does not give results as good as those obtained by working on the completed tree. So, due to lack of space, we are not showing these results here.

7. Experimental Evaluation of the Slim-tree

In this section we provide experimental results of the performance of the Slim-tree. We run experiments comparing the Slim-tree with the M-tree and demonstrating the impact of the MST-splitting method and the slim-down algorithm. The Slim-tree was implemented in C++ under Windows NT. The experiments were performed on a Pentium II 450MHz PC with 128 MB of main memory. Our implementation is based on a simple disk simulator which provides a counter for the number of disk accesses.

Since the performance of insertions is largely determined by the CPU-time (because of the required distance computations and the complexity of split operations), we report in the following only the total runtime for creating metric trees. For queries, however, we report the number of disk accesses which is a good indicator for the query performance. We found that the number of distance calculations is highly correlated with the number of disk accesses.

In our experiments we used the six previously introduced data sets. Let us mention here that the domain of the vector sets is the unit cube. The experiments were performed in the way that we first build up a metric tree by inserting tuples one by one. Thereafter, we run 13 sets of 500 range queries where the size of the range queries was fixed for each set. In the following graphs we report the average number of disk accesses obtained from a set of queries as the function of the query size. All the graphs are given in log-log scale.

7.1 Comparing the Slim-tree and the M-tree

Since the M-tree is the only dynamic metric tree available, we compared its alleged best performance to the corresponding one of our Slim-tree. Figure 9 shows the query performance of the Slim-tree and the M-tree for the six datasets. Both trees were built using the minMax-splitting algorithm. This algorithm was found to be the most promising for the M-tree [CPZ 97]. The corresponding capacities of the nodes used in these experiments are reported in Table 4. Note that for both trees we used the same settings of the parameters, leading to a fair comparison.

From the plots in Figure 9 we can see that the Slim-tree constantly outperforms M-tree. One of the reasons is that the occupation of the nodes is higher for the Slim-tree and therefore, the total number of nodes is smaller.

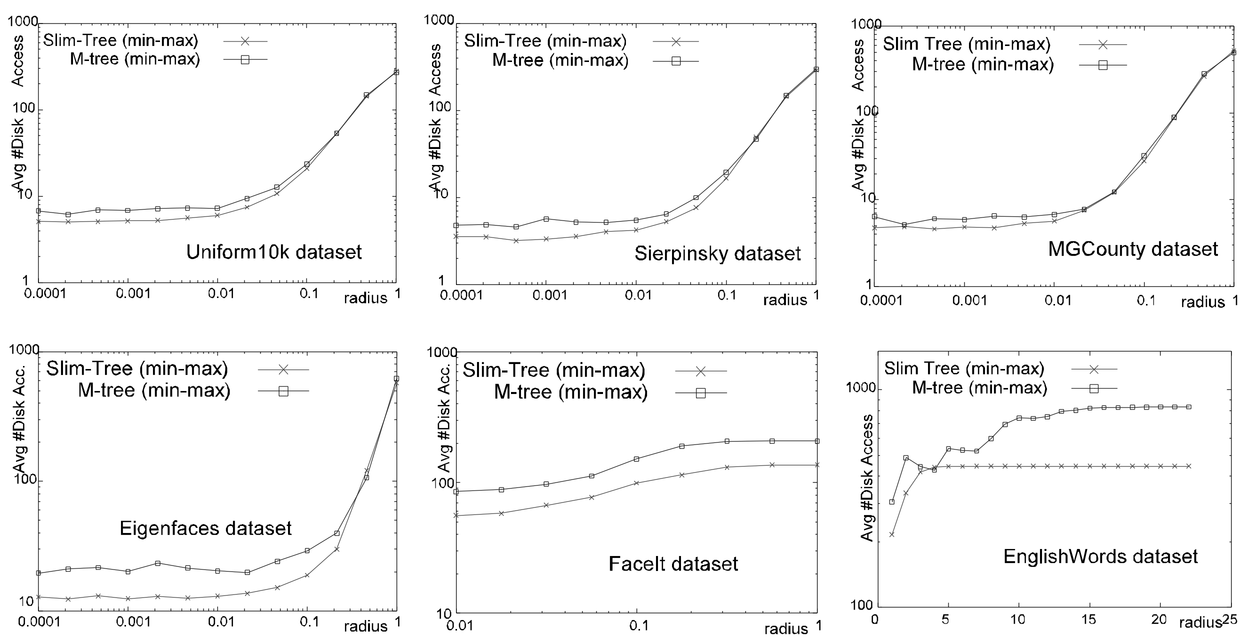


Figure 9. The query performance (given by average number of disk accesses) for the M-tree and the Slim-tree as a function of the query radius where each of the plots refers to one of our datasets.

This effect is visible for the FaceIt and EnglishWords datasets, where a large number of the pages are required for the large range queries. For the vector data sets, however, both trees perform similarly for large query radii. This is because the overlap of the entries is low and therefore, the different insertion strategies of the M-tree and the Slim-tree perform similarly. Note also that for large query radii it might be more effective to read the entire file into memory (using sequential I/Os). However it is common to expect that the majority of queries radii are rather small so that it is beneficial to use a metric tree.

7.2 - Comparing minMax and MST splitting algorithms

Figure 10 compares the query performance of two Slim-trees where the one uses the minMax-splitting algorithm and the other uses the MST-splitting algorithm. The left and right plot shows the results for Sierpinsky and FaceIt, respectively. The plot shows that both Slim-trees perform similarly. Table 3 give more details about the comparison of the different splitting strategies. Here, the columns “range queries” refer to the CPU time required to perform all the queries of the 13 sets. We point out here that the MST-splitting strategy suffers slightly when the number of objects per node (the capacity) is small. The columns “build” show the time to create the Slim-trees. The MST-algorithm is clearly superior to the minMax-splitting algorithm. For example,

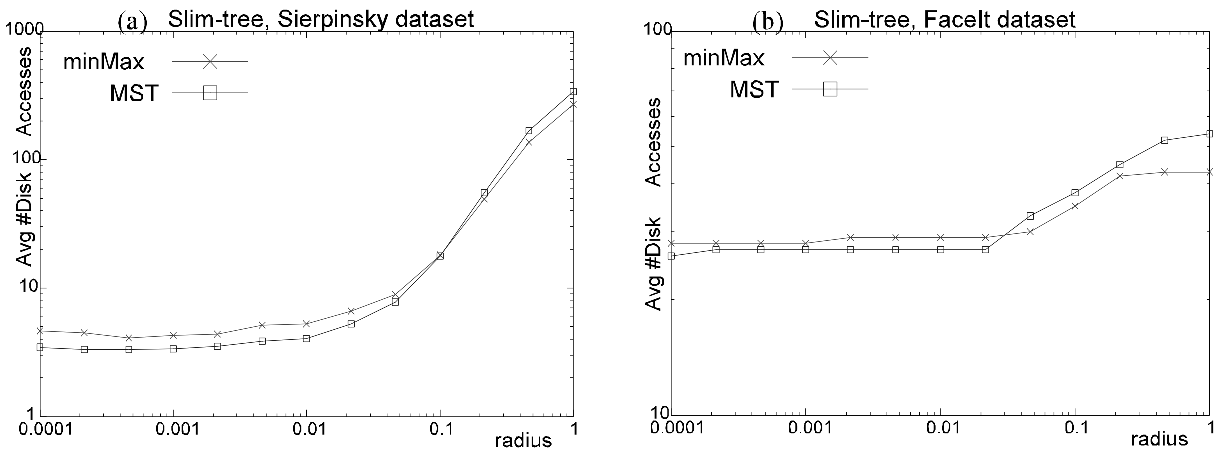


Figure 10. The query performance of two Slim-trees using the minMax-splitting algorithm and the MST-splitting algorithm. (a). Sierpinsky dataset, (b) FaceIt dataset.

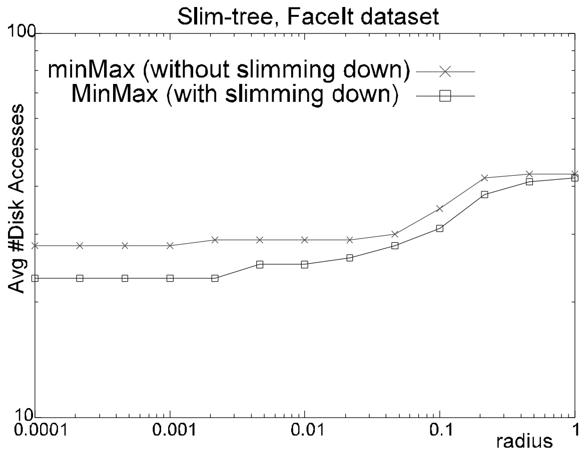
the MST-algorithm is faster by a factor of 60 for the two-dimensional datasets. Overall, the MST-splitting algorithm gives considerable savings when a Slim-tree is created and provides almost the same performance as the minMax-splitting algorithm for range queries.

The experiments on the splitting algorithms show that the runtime of the MST-splitting algorithm is increasingly better than the minMax-splitting algorithm as the number of entries increases. From the results of our experiments we can give the following rule of thumb for choosing a split algorithm: When *C* (the capacity of the nodes) is lower than 20, it is beneficial to use the minMax- splitting strategy. For trees with bigger *C*, the MST-splitting algorithm is a better choice. It is also important to say that the *ChooseSubtree* algorithm also has an influence on the splitting algorithms.

7.3 - Experiments with the Slim-down algorithm

The Slim-down algorithm improves the number of disk accesses for range queries in average between 10 to 20% for vector datasets. As these datasets already have a low bloat-factor, this gives small room for improvement through the Slim-down algorithm. For datasets with bigger bloat-factors (as the metric datasets FaceIt and

EnglishWords), the average improvement goes to between 25 and 35%.



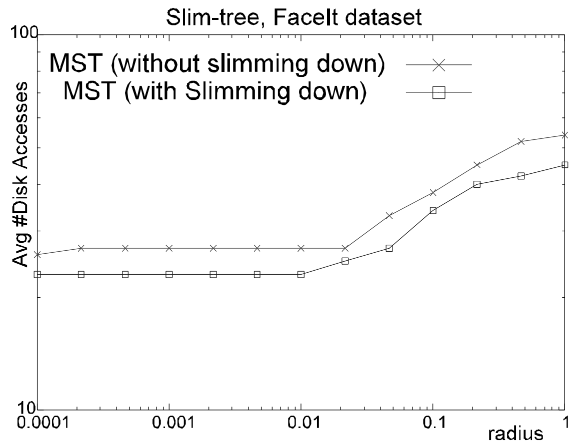


Figure 11. Comparing the improvements given by the Slim-down algorithm to answer range queries.

Figure 11 compares the query performance of the Slim-trees where the one uses the slim-down algorithm and the other does not. Note that on the left-hand side of Figure 11 we show the results when the minMax-splitting algorithm is used, whereas the results of the MST-splitting algorithm are presented on the right-hand side. Both graphs show that the slim-down algorithm improves the Slim-trees. In Table 3 (column "slim-down") we also show the time to perform the slim-down on the tree. In general, only a small fraction of the build time is required to perform a slim-down on a Slim-tree.

Datasets	Slim-tree using the minMax-splitting algorithm (time in sec.)				Slim-tree using the MST-splitting algorithm (time in sec.)			
	build	range queries	slim-down	get fat-factor	build	range queries	slim-down	get fat-factor
Uniform10k	94.62	17.23	0.23	1.54	1.58	16.62	0.33	1.91
Sierpinsky	103.88	11.86	0.41	0.83	1.68	11.96	0.23	0.99
MgCounty	195.16	20.67	0.38	2.31	3.08	20.78	0.40	2.16
Eigenfaces	61.60	62.33	1.20	0.13	6.46	68.67	0.89	71.10
EnglishWords	1704.44	504.17	82.59	173.36	33.96	513.03	210.41	204.78
FaceIt	0.45	0.13	0.08	0.06	0.15	0.16	0.01	0.16

Table 3 - A comparison of the Slim-trees using the minMax-splitting algorithm and the MST-splitting algorithm. The numbers are wall-clock times in seconds.

Although the measurement of I_C takes some computing time, the alternative way to obtain some values that represent the performance of a metric tree is to issue several queries for each given radius, keep the average number of disk accesses (or distance calculations) and their standard deviations and then generate the corresponding plots. We measured the times spent to calculate both the fat-factor and the average of disk accesses using 500 randomly generated queries. These measurements are shown in Table 3, and from there we can see that the calculation of the fat-factor is faster than the other alternative.

The last two columns of Table 4 show the fat-factor and the bloat-factor calculated for the Slim-trees using the MST-splitting algorithm. Moreover, we also present the number of nodes and the height of the tree. Note

Dataset	Num. of objects N	Objects per node C	Number of nodes M	M_{th}	Height of the tree H	H_{th}	Fat factor $Ff()$	Bloat factor $Bl()$
Uniform2D	10,000	52	307	198	3	3	0.03	0.05
Sierpinsky	9,841	52	374	195	3	3	0.01	0.01
MGCounty	15,559	52	568	307	3	3	0.01	0.01
Eigenfaces	11,900	24	930	518	4	3	0.32	0.58
FaceIt	1,056	11	226	106	4	3	0.23	0.51
EnglishWords	25,143	60	476	428	3	3	0.48	0.53

Table 4 - Parameters of the Slim-trees using the MST-splitting algorithm

that parameter M refers to the actual number of nodes and parameter M_{th} gives the minimum number of nodes. Analogously, the parameters H and H_{th} refer to the height of the Slim-tree. The number of objects per node (C) varies because of the size of the data objects.

In general, the results of our experiments confirmed that the fat-factor is suitable to measure the quality of a Slim-tree. As a rule of thumb, we believe that a metric tree with a fat-factor between 0 and 0.1 can be considered as a good tree, with a fat-factor between 0.1 and 0.4 as an acceptable tree, and with a fat-factor greater than 0.5 a bad tree. From Table 4 we can see that the Slim-trees for Uniform2D, Sierpinsky and MGCounty are very good trees, but the trees for Eigenfaces, FaceIt and EnglishWords are considered to be barely acceptable.

8. Conclusions

We have presented the Slim-tree, a dynamic metric access method which uses new approaches to efficiently index metric datasets. Our main contributions consist of the Slim-down algorithm and a new splitting algorithm based on the minimum spanning tree (MST). Additionally, we suggest a new *ChooseSubtree* algorithm for the Slim-tree which directs the insertion of an object from a given node to the child node with the lowest occupation when more than one child node qualifies. This leads to tighter trees and fewer disk pages, which also results in a more efficient processing of queries.

The new MST-splitting method is considerably faster than the minMax-splitting method, which has been considered the best for the M-tree [CPZ 97] [CP 98], while query performance is almost not effected. For a node with capacity C, the runtime of the minMax-splitting method is $O(C^3)$ whereas the runtime of the MST-splitting method is $O(C^2 \log C)$. Their performance difference is reflected in our experiments where the time to build a Slim-tree using the MST-splitting method, is by a factor of up two orders of magnitude lower than using the minMax-splitting method. Both splitting methods result in Slim-trees with almost the same query performance. We observed that query performance suffered a little by using the MST-splitting method, but only for small node capacities (less than 20).

The Slim-down algorithm is designed to be applied to a poorly constructed metric tree in order to improve its query performance. The theoretical underpinning of the Slim-down algorithm is our approach for computing overlap in a metric tree. Although overlap is identified as an important tuning parameter for improving query performance for spatial access methods, it has not been previously used for metric trees due to the inability to compute the volume of intersecting regions. In order to overcome this deficiency, we propose using the relative number of objects covered by two (or more) regions to estimate their overlap. This concept is used in the design of the Slim-down algorithm. In this paper, we used the Slim-down algorithm in a post-processing step, just after the insertion of all objects. This approach does not impedes subsequent insertions since it could also be used when objects have yet to be inserted. In our experiments, the Slim-down algorithm improves query performance by a factor of up to 35%.

Our concept of overlap also leads to the introduction of two factors each of them expresses the quality of a Slim-tree for a given dataset using only a single number. The fat-factor measures the quality of a tree with a fixed number of nodes, whereas the bloat factor enables a comparison of trees where the number of nodes is different. Moreover, we foresee that the proposed method of treating overlaps in metric spaces allows us to apply to metric access methods many well-known fine-tuning techniques developed for spatial access methods.

In our future work, we are primarily interested in the following subjects. First, we plan to apply the Slim-down algorithm to the nodes of the upper levels of a metric tree. So far, we have used the algorithm only for the internal nodes on the lowest level (the first level above the leaves). Second, we are interested in a comparison of the Slim-down algorithm with the re-insertion technique of the R*-tree. Third, we will study whether the bloat-factor can be used to develop a cost model for predicting the cost of a range query in the Slim-tree. Fourth, we will investigate whether the definition of the bloat-factor can be generalized when we explicitly distinguish between a set of query objects and a set of data objects.

Acknowledgments

We are grateful to Pavel Zezula, Paolo Ciaccia and Marco Patella for giving us the code of the M-tree.

References

[BCM 94] R. A. Baeza-Yates, W. Cunto, U. Manber, Sun Wu - *Proximity Matching Using Fixed-Queries Trees*, CPM: 198-212 (1994).

[BK 73] W. A. Burkhard, R.t M. Keller - *Some Approaches to Best-Match File Searching*, CACM 16(4): 230-236 (1973).

[BKS+ 90] N. Beckmann, H.P. Kriegel, R. Schneider and B. Seeger - The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", Proc. ACM-SIGMOD: 322-331 (1990).

[BBK+ 97] S. Berchtold, C. Böhm, D. A. Keim, H.-P. Kriegel - *A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space*. Proc. ACM-PODS 1997: 78-86.

[BO 97] T. Bozkaya, Z. M. Özsoyoglu - *Distance-Based Indexing for High-Dimensional Metric Spaces*, Proc. ACM-SIGMOD: 357-368 (1997).

[BRI 95] S. Brin - *Near Neighbor Search in Large Metric Spaces*, Proc. VLDB: 574-584 (1995).

[CPZ 97] P. Ciaccia, M. Patella, P. Zezula - *M-tree: An Efficient Access Method for Similarity Search in Metric Spaces*, Proc. VLDB: 426-435 (1997).

[CPZ 98] P. Ciaccia, M. Patella, F. Rabitti, P. Zezula - *Indexing Metric Spaces with M-tree*, Proc. Quinto convegno Nazionale SEBD, (1997).

[CP 98] P. Ciaccia, M. Patella - *Bulk Loading the M-tree*, Proc. ADC'98: 15-26 (1998).

[GG 98] V. Gaede, O. Gunther - "*Multidimensional Access Methods*", ACM Computing Surveys, Vol. 30, No. 2, 170-231, (1998).

[GUT 84] A. Guttman - "*R-Tree: Adynamic Index Structure for Spatial Searching*", Proc. ACM-SIGMOD Conference: 47-57 (1984).

[FK 94] C. Faloutsos, L. Kamel - *Beyond Uniformity and Independence: Analysis of R-tree Using the Concept of Fractal Dimension*, Proc. ACM-PODS: 4-13 (1994).

[KRU 56] J. B. Kruskal Jr. - *On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem*, Proc. Amer. Math. Soc., (7): 48-50 (1956).

[SRF 87] T. Sellis, N. Roussopoulos, C. Faloutsos - *The R+-tree: A Dynamic Index for Multi-dimensional*

Objects, Proc. VLDB: 507-518 (1987).

[TTF 99] C. Traina Jr., A. Traina, C. Faloutsos - *Distance Exponent: A New Concept for Selectivity Estimation in Metric Trees*, CMU-CS-99-110 Technical Report (1999).

[UHL 91] J. K. Uhlmann - *Satisfying General Proximity/Similarity Queries with Metric Trees*, IPL 40(4): 175-179 (1991).

[VIS 98] Visionics Corp. - Available at http://www.visionics.com/live/frameset.html (12-Feb-1999).

[WTS+ 96] H. D. Wactlar, T. Kanade, M. A. Smith and S. M. Stevens - *"Intelligent Access to Digital Video: Informedia Project"* , IEEE Computer, vol. 29 (3): 46-52 (1996).

[YIA 93] P. N. Yianilos - *Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces*, SODA: 311-321 (1993).