Pivoting M-tree: A Metric Access Method for Efficient Similarity Search

Tomáš Skopal

Department of Computer Science, VŠB–Technical University of Ostrava,
tř. 17. listopadu 15, Ostrava, Czech Republic
tomas.skopal@vsb.cz

Abstract. In this paper pivoting M-tree (PM-tree) is introduced, a metric access method combining M-tree with the pivot-based approach. While in M-tree a metric region is represented by a hyper-sphere, in PM-tree the shape of a metric region is determined as an intersection of the hyper-sphere and a set of hyper-rings. The set of hyper-rings for each metric region is related to a fixed set of pivot objects. As a consequence, the shape of a metric region bounds the indexed objects more tightly which, in turn, improves the overall efficiency of the similarity search. Preliminary experimental results on a synthetic dataset are included.

Keywords: PM-tree, M-tree, pivot-based methods, efficient similarity search

1 Introduction

Together with the increasing volume of various multimedia collections, the need for an efficient similarity search in large multimedia databases becomes stronger. A multimedia document (its main features respectively) is modelled by an object (usually a vector) in a feature space $\mathcal{U}$ thus the whole collection can be represented as a dataset $S \subset \mathcal{U}$. Similarity search is then provided using a *spatial access method* [1] which should efficiently retrieve those objects from the dataset that are relevant to a given similarity query.

In context of similarity search, a similarity function (dissimilarity function actually) can be modeled using a metric, i.e. a distance function d satisfying the following metric axioms for all $O_i, O_j, O_k \in \mathcal{U}$:

$$\begin{aligned} d(O_i, O_i) &= 0 && && \text{reflexivity} \\ d(O_i, O_j) &> 0 && (O_i \neq O_j) && \text{positivity} \\ d(O_i, O_j) &= d(O_j, O_i) && && \text{symmetry} \\ d(O_i, O_j) + d(O_j, O_k) &\geq d(O_i, O_k) && && \text{triangular inequality} \end{aligned}$$

Given a metric space $\mathcal{M} = (\mathcal{U}, d)$, the *metric access methods* [2] organize (or index) objects of a dataset $S \subset \mathcal{U}$ just using the metric d. Most of the metric access methods exploit a structure of metric regions within the space $\mathcal{M}$. Common to all these methods is that during a search process the triangular inequality of d allows to discard some irrelevant subparts of the metric structure.

© V. Snášel, J. Pokorný, K. Richta (Eds.): Dateso 2004, pp. 27–37, ISBN 80-248-0457-3.
VŠB – Technical University of Ostrava, Dept. of Computer Science, 2004.

2 M-tree

Among many of metric access methods developed so far, the M-tree [3,5] (and its modifications) remains still the only indexing technique suitable for an efficient similarity search in large multimedia databases.

The M-tree is based on a hierarchical organization of feature objects $O_i \in S$ according to a given metric d. Like other dynamic, paged trees, the M-tree structure is a balanced hierarchy of nodes. The nodes have a fixed capacity and a utilization threshold. Within the M-tree hierarchy, the objects are clustered into metric regions. The leaf nodes contain *ground entries* of indexed objects themselves while *routing entries* (stored in the inner nodes) represent the metric regions. A ground entry has a format:

$$grnd(O_i) = [O_i, oid(O_i), d(O_i, P(O_i))]$$

where $O_i \in S$ is an appropriate feature object, $oid(O_i)$ is an identifier of the original DB object (stored externally), and $d(O_i, P(O_i))$ is a precomputed distance between O_i and its parent routing entry. A routing entry has a format:

$$rout(O_j) = [O_j, ptr(T(O_j)), r(O_j), d(O_j, P(O_j))]$$

where $O_j \in S$ is a feature object, $ptr(T(O_j))$ is pointer to a covering subtree, $r(O_j)$ is a covering radius, and $d(O_j, P(O_j))$ is a precomputed distance between O_j and its parent routing entry (this value is zero for the routing entries stored in the root). The routing entry determines a hyper-spherical metric region in space $\mathcal{M}$ where the object O_j is a center of that region and $r(O_j)$ is a radius bounding the region. The precomputed value $d(O_j, P(O_j))$ is redundant and serves for optimizing the M-tree algorithms.

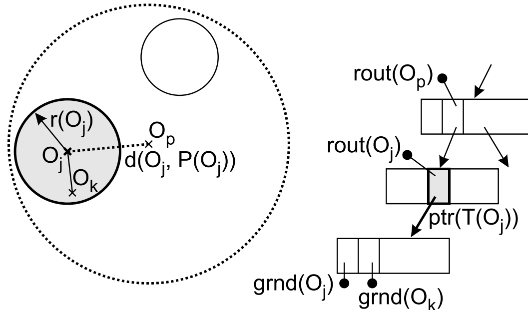

Fig. 1. A metric region and its routing entry in the M-tree structure.

In Figure 1, a metric region and its appropriate routing entry $rout(O_j)$ in an M-tree are presented. For a hierarchy of metric regions (routing entries $rout(O_j)$ respectively) the following condition must be satisfied:

All feature objects stored in leafs of covering subtree of $rout(O_j)$ must be spatially located inside the region defined by $rout(O_j)$.

Formally, having a $rout(O_j)$ then $\forall O_i \in T(O_j), d(O_i, O_j) \leq r(O_j)$. If we realize, such a condition is very weak since there can be constructed many M-trees of the same object content but of different structure. The most important consequence is that many regions on the same M-tree level may overlap.

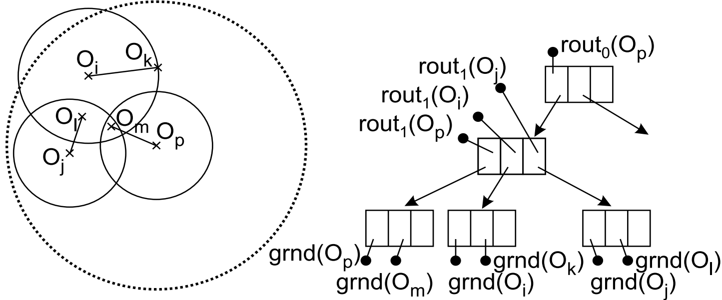

Fig. 2. Hierarchy of metric regions and the appropriate M-tree.

An example in Figure 2 shows several objects partitioned into metric regions and the appropriate M-tree. We can see that the regions defined by $rout_1(O_p)$, $rout_1(O_i)$, $rout_1(O_j)$ overlap. Moreover, object O_l is located inside the regions of $rout_1(O_i)$ and $rout_1(O_j)$ but it is stored just in the subtree of $rout_1(O_j)$. Similarly, the object O_m is located even in three regions but it is stored just in the subtree of $rout_1(O_p)$.

2.1 Similarity Queries

The structure of M-tree was designed to natively support similarity queries. A similarity measure is here represented by the metric function d. Given a query object O_q, a similarity query returns (in general) objects $O_i \in S$ close to O_q.

In the context of similarity search we distinguish two kinds of queries. A *range query* is specified as a hyper-spherical *query region* defined by a query object O_q and a query radius $r(O_q)$. The purpose of a range query is to return all the objects $O_i \in S$ satisfying $d(O_q, O_i) \leq r(O_q)$. A query with $r(O_q) = 0$ is called a *point query*. A *k-nearest neighbours query* (k-NN query) is specified by a query object O_q and a number k. A k-NN query returns the first k nearest objects to O_q. Technically, a k-NN query can be implemented using a range query with a dynamic query radius.

During a similarity query processing the M-tree hierarchy is being traversed down. Only if a routing object $rout(O_j)$ (its metric region respectively) intersects the query region, the covering subtree of $rout(O_j)$ is relevant to the query and thus further processed.

2.2 Retrieval Efficiency

The retrieval efficiency of an M-tree (i.e. the costs of a query evaluation) is highly dependent on the amount of overall volume[1] of the metric regions described by routing entries. The larger metric region volumes (and also volumes of region overlaps) the higher probability of intersection with a query region.

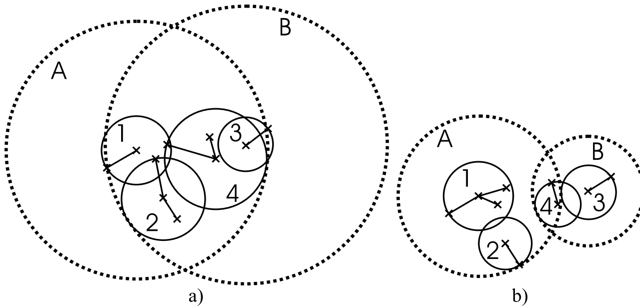

Fig. 3. a) An M-tree with large volume of regions. b) An M-tree with small volume of regions.

In Figure 3 two different yet correct M-tree hierarchies for the same dataset are presented. Although both M-trees organize the same dataset, a query processing realized on the second M-tree will be more efficient (in average) due to the smaller region volumes.

Recently, we have introduced two algorithms [6] leading to a reduction of the overall volume of metric regions. The first method, the multi-way dynamic insertion, finds the most appropriate leaf for each object being inserted. The second (post-processing) method, the generalized slim-down algorithm, "horizontally" (i.e. separately for each tree level) tries to redistribute all entries among more appropriate nodes.

3 Pivoting M-tree

A metric region (as a part of routing entry) of M-tree is described by a bounding hyper-sphere (given by a center object and a radius). However, the shape of hyper-spherical region is far from optimal since it does not "wrap" the objects tightly together and the region volume is too large. In other words, relatively to the hyper-sphere volume there is only a "few" objects spread inside the hyper-sphere thus a huge proportion of an empty space[2] is covered. Consequently,

[1] We consider only an imaginary volume since there exists no universal notion of volume in general metric spaces.

[2] The uselessly indexed empty space is sometimes refered as a "dead space".

for hyper-spherical regions of large volumes the query processing becomes less efficient.

In this section we introduce an extension of M-tree, called *pivoting M-tree* (PM-tree), exploiting the pivod-based idea for metric region volumes reduction.

3.1 Pivot-based Methods

Similarity search realized by pivot-based methods [2,4] is based on a single general idea. A set of p (random) objects $\{p_1, ..., p_l, ..., p_k\} \subset S$ is selected, called *pivots*. The dataset S (of size n) is preprocessed so as to build a table of $n * p$ entries, where all the distances $d(O_i, p_l)$ are stored for every $O_i \in S$ and every pivot p_l. When a range query $(O_q, r(O_q))$ is processed, we compute $d(O_q, p_l)$ for every pivot p_j and then try to discard such O_i that $|d(O_i, p_l) - d(O_q, p_l)| > r(O_q)$. The objects O_i which cannot be eliminated with this rule have to be directly compared against O_q.

The simple pivot-based approach is suitable especially for applications where the distance d is considered expensive to compute. However, it is obvious that the whole table of $n*p$ entries must be sequentially loaded during a query processing which significantly increases the disk access costs.

3.2 Structure of PM-tree

Since PM-tree is an extension of M-tree we just describe the new facts instead of a comprehensive definition. To exploit advantages of both, the M-tree and the pivot-based approach, we have enhanced the routing and ground entries by a pivot-based information.

First of all, a set of p pivots $p_l \in S$ must be selected. This set is fixed for all the lifetime of a particular PM-tree index. Furthermore, we define a routing entry of a PM-tree inner node as:

$$rout_{PM}(O_j) = [O_j, ptr(T(O_j)), r(O_j), d(O_j, P(O_j)), HR]$$

The additional HR attribute stands for an array of p_{hr} *hyper-rings* ($p_{hr} \leq p$) where the l-th hyper-ring $HR[l]$ is an interval (possibly the smallest) covering distances between the pivot p_l and each of the objects stored in leafs of $T(O_j)$, i.e. $HR[l].min = min(\{d(O_i, p_l)\})$ and $HR[l].max = max(\{d(O_i, p_l)\})$ for $\forall O_i \in T(O_j)$. Similarly, for a PM-tree leaf we define a ground entry as:

$$grnd_{PM}(O_i) = [O_i, oid(O_i), d(O_i, P(O_i)), PD]$$

The additional PD stands for an array of p_{pd} *pivot distances* ($p_{pd} \leq p$) where the l-th distance $PD[l] = d(O_i, p_l)$.

Since each hyper-ring stored in HR defines a metric region containing *all* the objects indexed by $T(O_j)$, an intersection of hyper-rings and the hyper-sphere forms a metric region bounding all the objects in $T(O_j)$. Furthermore, due to the intersection with hyper-sphere, the PM-tree metric region is always smaller

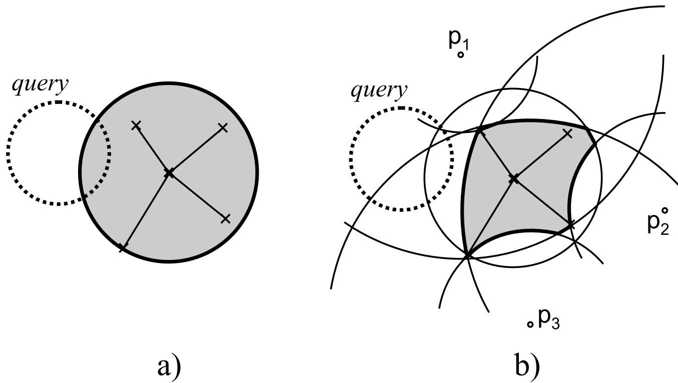

Fig. 4. a) Region of M-tree. b) Reduced region of PM-tree (with three pivots).

than the original M-tree region defined just by a hyper-sphere. For a comparison of an M-tree region and an equivalent PM-tree region see Figure 4.

The PM-tree, as a combination of M-tree and the idea of pivoting, represents a metric access method based on *hierarchical pivoting*. The numbers p_{hr} and p_{pd} (both fixed during a PM-tree index lifetime) allow us to specify the "amount of pivoting". For $p_{hr} > 0$ and $p_{pd} = 0$ only the hierarchical pivoting will take place while for $p_{hr} = 0$ and $p_{pd} > 0$ a query will be processed like in the ordinary M-tree with subsequent pivot-based filtering in leafs. Obviously, using a suitable $p_{hr} > 0$ and $p_{pd} > 0$ the PM-tree can be tuned to achieve an optimal storage/retrieval efficiency.

3.3 Building the PM-tree

In order to keep *HR* and *PD* arrays up-to-date, the original M-tree construction algorithms [5,6] must be adjusted. The adjusted algorithms still preserve the logarithmic time complexity.

Object Insertion.
During an object O_i insertion, the *HR* array of each routing entry in the insertion path must be updated by values $d(O_i, p_l), \forall l \leq p_{hr}$.

For the leaf node in the insertion path a new ground entry must be created together with filling its *PD* array by values $d(O_i, p_l), \forall l \leq p_{pd}$.

Node Splitting.
When a node is split, a new *HR* array of the left new routing entry is created by union of all appropriate intervals $HR[l]$ ($PD[l]$ in case of leaf splitting) stored in routing entries (ground entries respectively) of the left new node. A new *HR* array of the right new routing entry is created similarly.

3.4 Query Processing

Before processing any similarity query the distances $d(O_q, p_l), \forall l \leq max(p_{hr}, p_{pd})$ have to be computed. During a query processing the PM-tree hierarchy is being traversed down. Only if the metric region of a routing entry $rout(O_j)$ intersects the query region $(O_q, r(O_q))$, the covering subtree $T(O_j)$ may be relevant to the query and thus it is further processed. In case of a relevant PM-tree routing entry the query region must intersect all the hyper-rings stored in *HR*. Prior to the standard hyper-sphere intersection check (used by M-tree), the intersection of hyper-rings $HR[l]$ with the query region is checked as follows (note that no additional d computation is needed):

$$\bigwedge_{l=1}^{p_{hr}} (d(O_q, p_l) - r(O_q) \leq HR[l].max \ \wedge \ d(O_q, p_l) + r(O_q) \geq HR[l].min)$$

If the above hyper-ring intersection condition is false, the subtree $T(O_j)$ is irrelevant to the query and thus discarded from further processing. On the leaf level a relevant ground entry is determined such that the following condition must be satisfied:

$$\bigwedge_{l=1}^{p_{pd}} |d(O_q, p_l) - PD[l]| \leq r(O_q)$$

In Figure 4 an example of query processing is presented. Although the M-tree metric region cannot be discarded (see Figure 4a), the PM-tree region can be discarded since the hyper-ring $HR[2]$ is not intersected (see Figure 4b).

The hyper-ring intersection condition can be incorporated into the original M-tree range query as well as k-NN query algorithms. In case of range query the adjustment is straightforward – the hyper-ring intersection condition is combined with the original hyper-sphere intersection condition. However, the k-NN query algorithm (based on priority queue heuristics) must be redesigned. In the experiments we have considered range queries only – the design of a k-NN query algorithm for PM-tree is a subject of our future research.

3.5 Hyper-Ring Storage

In order to minimize storage volume of the *HR* and *PD* arrays in PM-tree nodes, a short representation of object-to-pivot distance is necessary.

We can represent a hyper-ring $HR[l]$ by two 4-byte reals and a pivot distance $PD[l]$ by one 4-byte real. When (a part of) the dataset is known in advance we can approximate the 4-byte distance representation by a 1-byte code. For this reason a distance distribution histogram is created by random sampling of objects from the dataset along with comparing them against all the pivots. Then a distance interval $\langle d_{min}, d_{max} \rangle$ is computed so that most of the histogram distances fall into the interval. See an example in Figure 5, where such an interval covers 90% of sampled distances (the d^+ value is an (estimated) maximum distance of a bounded metric space $\mathcal{M}$).

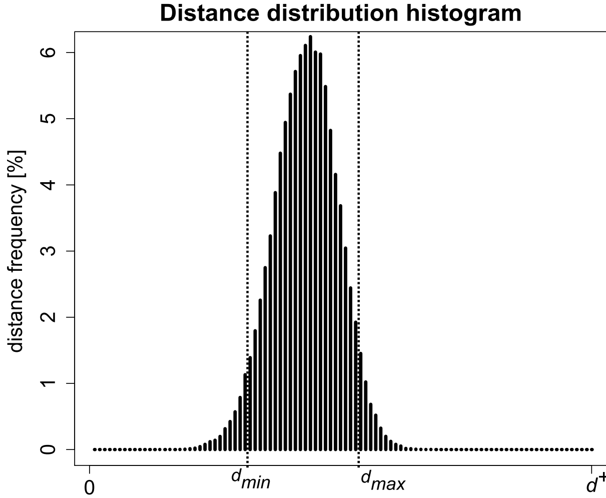

Fig. 5. Distance distribution histogram, 90% distances in interval $\langle d_{min}, d_{max} \rangle$

Values $HR[l]$ and $PD[l]$ are scaled into the $\langle d_{min}, d_{max} \rangle$ interval using a 1-byte code. Experimental results have shown that a 1-byte distance approximation is almost as effective as a 4-byte real while by using 1-byte approximation the PM-tree storage savings are considerable. As an example, for $p_{hr} = 50$ together with using 4-byte distances, the hyper-rings stored in an inner node having capacity 30 entries will consume $30 * 50 * 2 * 4 = 12000$ bytes while by using 1-byte distance codes the hyper-rings will take only $30 * 50 * 2 * 1 = 3000$ bytes.

4 Experimantal Results

We have made several preliminary experiments on a synthetic dataset of 250,000 10-dimensional vectors. The vectors were distributed within 2500 spherical clusters of a fixed radius (over the whole extent of the vector space domain). As a distance function the Euclidean metric (L_2) was used. Each label `PM-tree(x,y)` in the figures below stands for a PM-tree index where p_{hr} = x and p_{pd} = y. The sizes of PM-tree indices varied from 19MB (in case of PM-tree (0,0), i.e. M-tree) to 64MB (in case of PM-tree (100,100)). The PM-tree node size (disk page size respectively) was set to 4KB. For each index construction the `SingleWay + MinMax` techniques were used (we refer to [6]).

In the experiments a retrieval efficiency of range query processing was evaluated. The query objects were randomly selected from the dataset and each particular query test consisted of 200 range queries of the same query selectivity (the number of objects in query result). The results were averaged. Disk access costs (DAC) and computation costs of the query evaluation were examined, according to the number of pivots used (p_{hr} and/or p_{pd}) as well as according to

query selectivity. The query selectivity was ranged from 5 to 50 objects. The experiments were intended to compare PM-tree with M-tree hence the PM-tree query costs are related to the costs spent by processing the same query by the M-tree index.

4.1 Disk Access Costs

In Figure 6a DAC according to query selectivity are presented. We can see that for querying `PM-tree(60,0)` index there is needed from 80% to 90% (increasing with selectivity) of DAC needed by the M-tree. The `PM-tree(200,0)` index is even more efficient since only 65% to 85% of DAC is needed. On the other side, `PM-tree(200,50)` index consumes up to 150% DAC since the long *PD* arrays (storing 50 pivot distances for each ground entry) cause the 250,000 ground entries must be stored in 9177 leafs (the M-tree needs only 4623 leafs).

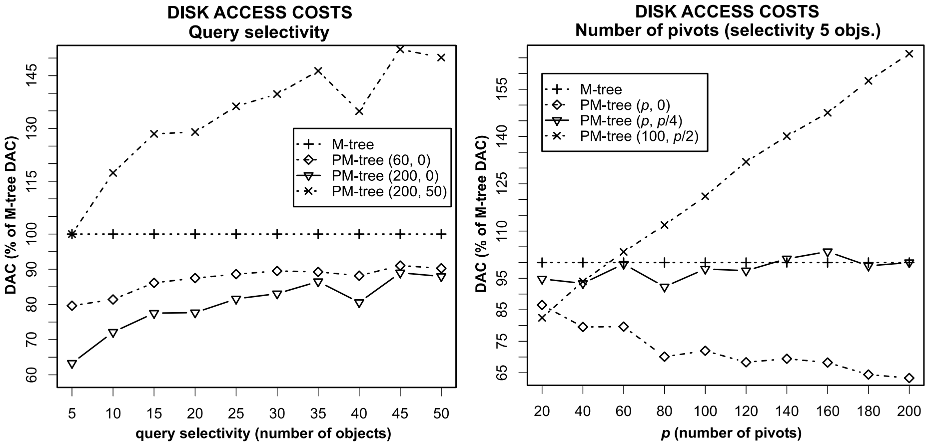

Fig. 6. Disk access costs: a) Query selectivity b) Number of pivots

Disk access costs according to the number of pivots are presented in Figure 6b. With the increasing p the disk access costs for PM-tree(p,0) indices decrease from 85% to 65% since more hyper-rings help to discard more irrelevant subtrees while the index sizes grow slowly (e.g. size of `PM-tree(200,0)` index is 24MB). The PM-tree(100,$p/2$) indices are more efficient than the M-tree for $p < 60$ only.

Interesting results are presented for PM-tree(p,$p/4$) indices where DAC remain about 100%. These results are better than for PM-tree(100,$p/2$) indices but worse than for PM-tree(p,0) indices. The reason for such behaviour is that

with increasing p more hyper-rings help to discard more irrelevant subtrees but, on the other hand, due to the even longer PD arrays the index sizes grow quickly.

4.2 Computation Costs

Unlike for disk access costs, the increasing number of pivot distances in PD arrays positively affects the computation costs. In Figure 7a we can observe `PM-tree(200,50)` index to be more than 10 times as efficient as the M-tree index. However, for $p > 80$ the indices `PM-tree(`p`,`$p/4$`)` and `PM-tree(100,`$p/2$`)` consume the same computation costs (see Figure 7b). This happens particularly due to the increasing number of leafs which must be payed by a higher number of routing entries in inner nodes.

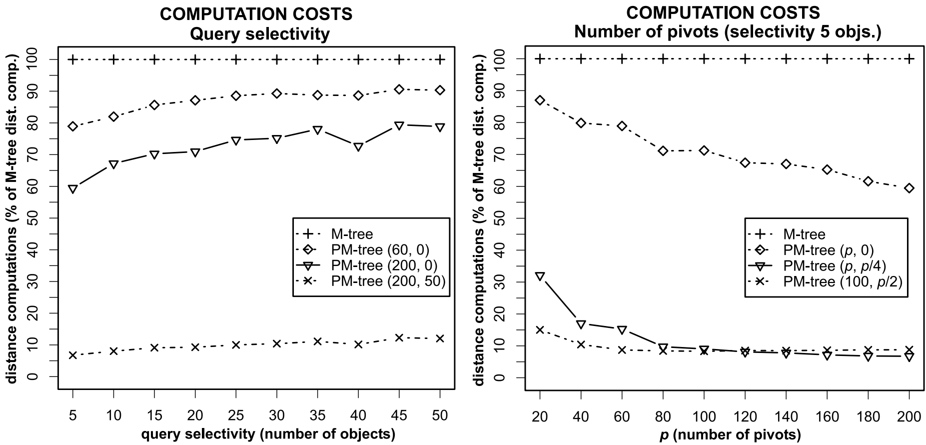

Fig. 7. Computation costs: a) Query selectivity b) Number of pivots

4.3 Summary

Based on the experimental results we are able to claim several facts (relative to the M-tree efficiency):

- For increasing p where $p_{hr} = p$ and $p_{pd} = 0$ the disk access costs as well as the computation costs steadily decrease.
- For increasing p where $p_{hr} \gg p_{pd}$ (say $p_{hr} = p, p_{pd} = \frac{p}{4}$) the disk access costs are similar to the M-tree DAC but the computation costs can be considerably lower. Such a behaviour can be useful when a distance computation is more expensive than a single disk access.
- In cases where $p_{hr} \leq p_{pd}$ the PM-tree behaviour acts similarly like the simple pivot-based filtering does since the disk access costs are high.

5 Conclusions and Outlook

In this paper the pivoting M-tree (PM-tree) was introduced. The PM-tree combines M-tree hierarchy of metric regions together with the idea of pivot-based methods. The result is a flexible metric access method providing even more efficient similarity search than the M-tree. The preliminary experimental results on a synthetic dataset indicate various efficiency trends for various PM-tree configurations.

In the future we plan to develop new PM-tree building algorithms exploiting the pivot-based information. Second, the original M-tree k-NN query algorithm has to be redesigned. Our next goal is formulation of a cost model making possible to tune PM-tree parameters for an estimated efficiency. Last but not least, extensive experiments on huge multimedia datasets have to be performed.

References

1. 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, 2001.
2. E. Chávez, G. Navarro, R. Baeza-Yates, and J. Marroquín. Searching in Metric Spaces. *ACM Computing Surveys*, 33(3):273–321, 2001.
3. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of the 23rd Athens Intern. Conf. on VLDB*, pages 426–435. Morgan Kaufmann, 1997.
4. L. Mico, J. Oncina, and E. Vidal. A new version of the nearest-neighbor approximating and eliminating search (aesa) with linear preprocessing-time and memory requirements. *Pattern Recognition Letters*, 15:9–17, 1994.
5. M. Patella. *Similarity Search in Multimedia Databases*. Dipartmento di Elettronica Informatica e Sistemistica, Bologna, 1999.
6. T. Skopal, J. Pokorný, M. Krátký, and V. Snášel. Revisiting M-tree Building Principles. In *ADBIS 2003, LNCS 2798, Springer-Verlag, Dresden, Germany*, 2003.