Improving the Performance of M-tree Family by Nearest-Neighbor Graphs

Tomáš Skopal, David Hoksza

Charles University in Prague, FMP, Department of Software Engineering
Malostranské nám. 25, 118 00 Prague, Czech Republic
{tomas.skopal, david.hoksza}@mff.cuni.cz

Abstract. The M-tree and its variants have been proved to provide an efficient similarity search in database environments. In order to further improve their performance, in this paper we propose an extension of the M-tree family, which makes use of nearest-neighbor (NN) graphs. Each tree node maintains its own NN-graph, a structure that stores for each node entry a reference (and distance) to its nearest neighbor, considering just entries of the node. The NN-graph can be used to improve filtering of non-relevant subtrees when searching (or inserting new data). The filtering is based on using "sacrifices" – selected entries in the node serving as pivots to all entries being their reverse nearest neighbors (RNNs). We propose several heuristics for sacrifice selection; modified insertion; range and kNN query algorithms. The experiments have shown the M-tree (and variants) enhanced by NN-graphs can perform significantly faster, while keeping the construction cheap.

1 Introduction

In multimedia retrieval, we usually need to retrieve objects based on similarity to a query object. In order to retrieve the relevant objects in an *efficient* (quick) way, the similarity search applications often use *metric access methods* (MAMs) [15], where the similarity measure is modeled by a metric distance δ. The principle of all MAMs is to employ the triangle inequality satisfied by any metric, in order to partition the dataset $\mathbb{S}$ among classes (organized in a *metric index*). When a query is to be processed, the metric index is used to quickly filter the non-relevant objects of the dataset, so that the number of distance computations needed to answer a query is minimized.[1] In the last two decades, there were many MAMs developed, e.g., gh-tree, GNAT, (m)vp-tree, SAT, (L)AESA, D-index, M-tree, to name a few (we refer to monograph [15]). Among them, the M-tree (and variants) has been proved as the most universal and suitable for practical database applications.

In this paper we propose an extension to the family of M-tree variants (implemented for M-tree and PM-tree), which is based on maintaining an additional structure in each tree node – the nearest-neighbor (NN) graph. We show that utilization of NN-graphs can speed up the search significantly.

[1] The metric is often supposed expensive, so the number of distance computations is regarded as the most expensive component of the overall runtime costs.

2 M-tree

The *M-tree* [7] is a dynamic (easily updatable) index structure that provides good performance in secondary memory (i.e. in database environments). The M-tree index is a hierarchical structure, where some of the data objects are selected as centers (references or local *pivots*) of ball-shaped regions, and the remaining objects are partitioned among the regions in order to build up a balanced and compact hierarchy of data regions, see Figure 1a. Each region (subtree) is indexed recursively in a B-tree-like (bottom-up) way of construction.

The inner nodes of M-tree store *routing entries*

$$rout_l(O_i) = [O_i, r_{O_i}, \delta(O_i, Par(O_i)), ptr(T(O_i))]$$

where $O_i \in \mathbb{S}$ is a data object representing the center of the respective ball region, r_{O_i} is a *covering radius* of the ball, $\delta(O_i, Par(O_i))$ is so-called *to-parent* distance (the distance from O_i to the object of the parent routing entry), and finally $ptr(T(O_i))$ is a pointer to the entry's subtree. The data is stored in the leaves of M-tree. Each leaf contains *ground entries*

$$grnd(O_i) = [O_i, \delta(O_i, Par(O_i))]$$

where $O_i \in \mathbb{S}$ is the indexed data object itself, and $\delta(O_i, Par(O_i))$ is, again, the to-parent distance. See an example of routing and ground entries in Figure 1a.

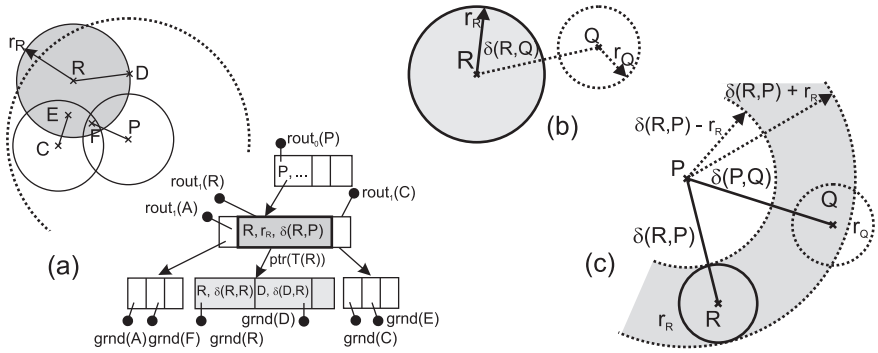

Fig. 1. (a) Example of an M-tree (b) Basic filtering (c) Parent filtering

2.1 Query processing

The range and k nearest neighbors (kNN) queries are implemented by traversing the tree, starting from the root[2]. Those nodes are accessed, the parent region (R, r_R) of which (described by the routing entry) is overlapped by the query ball (Q, r_Q). In case of a kNN query (we search for k closest objects to Q), the radius r_Q is not known in advance, so we have to additionally employ a heuristic to dynamically decrease the radius during the search (initially set to ∞).

[2] We just outline the main principles, the algorithms are described in Section 4, including the proposed extensions. The original M-tree algorithms can be found in [7].

Basic filtering. The check for region-and-query overlap requires an explicit distance computation $\delta(R, Q)$, see Figure 1b. In particular, if $\delta(R, Q) \leq r_Q + r_R$, the data ball R overlaps the query ball, thus the child node has to be accessed. If not, the respective subtree is filtered from further processing.

Parent filtering. As each node in the tree contains the distances from the routing/ground entries to the center of its parent node, some of the non-relevant M-tree branches can be filtered out without the need of a distance computation, thus avoiding the "more expensive" basic overlap check (see Figure 1c). In particular, if $|\delta(P, Q) - \delta(P, R)| > r_Q + r_R$, the data ball R cannot overlap the query ball, thus the child node has not to be re-checked by basic filtering. Note $\delta(P, Q)$ was computed in the previous (unsuccessful) parent's basic filtering.

2.2 M-tree Construction

Starting at the root, a new object O_i is recursively inserted into the best subtree $T(O_j)$, which is defined as the one for which the covering radius r_{O_j} must increase the least in order to cover the new object. In case of ties, the subtree whose center is closest to O_i is selected. The insertion algorithm proceeds recursively until a leaf is reached and O_i is inserted into that leaf. A node's overflow is managed in a similar way as in the B-tree – two objects from the overflowed node are selected as new centers, the node is split, and the two new centers (forming two routing entries) are promoted to the parent node. If the parent overflows, the procedure is repeated. If the root overflows, it is split and a new root is created.

3 Related work

In the last decade, there have been several modifications/successors of M-tree introduced, some improving the performance of M-tree, some others improving the performance but restricting the general metric case into vector spaces, and, finally, some adjusting M-tree for an extended querying model.

In the first case, the *Slim-tree* [14] introduced two features. First, a new policy of node splitting using the minimum spanning tree was proposed to reduce internal CPU costs during insertion (but not the number of distance computations). The second feature was the slim-down algorithm, a post-processing technique for redistribution of ground entries in order to reduce the volume of bottom-level data regions. Another paper [12] extended the slim-down algorithm to the general case, where also the routing entries are redistributed. The authors of [12] also introduced the multi-way insertion of a new object, where an optimal leaf node for the object is found. A major improvement in search efficiency has been achieved by the proposal of *PM-tree* [13, 10] (see Section 5.1), where the M-tree was combined with pivot-based techniques (like LAESA).

The second case is represented by the *M^+-tree* [16], where the nodes are further partitioned by a hyper-plane (where a key dimension is used to isometrically partition the space). Because of hyper-plane partitioning, the usage of M^+-tree

is limited just to Euclidean vector spaces. The *BM$^+$-tree* [17] is a generalization of M$^+$-tree where the hyper-plane can be rotated, i.e. it has not to be parallel to the coordinate axes (the restriction to Euclidean spaces remains).

The last category is represented by the *M^2-tree* [5], where the M-tree is generalized to be used with an arbitrary aggregation of multiple metrics. Another structure is the *M^3-tree* [3], a similar generalization of M-tree as the M^2-tree, but restricted just to linear combinations of partial metrics, which allows to effectively compute the lower/upper bounds when using query-weighted distance functions. The last M-tree modification of this kind is the *QIC-M-tree* [6], where a user-defined query distance (even non-metric) is supported, provided a lower-bounding metric to the query distance (needed for indexing) is given by user.

In this paper we propose an extension belonging to the first category (i.e. improving query performance of the general case), but which could be utilized in all the M-tree modifications (M-tree family) overviewed in this section.

4 M*-tree

We propose the M*-tree, an extension of M-tree having each node additionally equipped by *nearest-neighbor graph* (NN-graph). The motivation for employing NN-graphs is related to the advantages of methods using *global pivots* (objects which all the objects in dataset are referenced to). Usually, the global pivots are used to filter out some non-relevant objects from the dataset (e.g. in LAESA [9]). In general, the global pivots are good if they are far from each other and provide different "viewpoints" with respect to the dataset [2]. The M-tree, on the other hand, is an example of a method using *local pivots*, where the local pivots are the parent routing entries of nodes. As a hybrid structure, the PM-tree is combining the "local-pivoting strategies" of M-tree with the "global-pivoting strategies" of LAESA.

We can also state a "closeness criterion" for good pivots, as follows. A pivot is good in case it is close to a data object or, even better, close to the query object. In both cases, a close pivot provides tight distance approximation to the data/query object, so that filtering of data object by use of precomputed pivot-to-query or pivot-to-data distances is more effective. Unfortunately, this criterion cannot be effectively utilized for global pivots, because there is only a limited number of global pivots, while there are many objects referenced to them (so once we move a global pivot to become close to an object, many other objects become handicapped). Nevertheless, the closeness criterion can be utilized in local-pivoting strategies, because only a limited number of dataset objects are referenced to a local pivot and, conversely, a dataset object is referenced to a small number of pivots. Hence, a replacement of local pivot would impact only a fraction of objects within the entire dataset.

4.1 Nearest-neighbor graphs

In M-tree node, there exists just one local pivot, the parent (which moreover plays the role of center of the region). The parent filtering described in Section 2.1

is, actually, pivot-based filtering where the distance to the pivot is precomputed so we can avoid computation of the query-to-object distance (see Figure 1c). However, from the closeness criterion point of view, the parent is not guaranteed to be close to a particular object in the node, it is rather a compromise which is "close to all of them".

Following the closeness criterion, in this paper we propose a technique where all the objects in an M-tree node mutually play the roles of local pivots. In order to reduce space and computation costs, each object in a node is explicitly referenced just to its nearest neighbor. This fits the closeness criterion; we obtain a close pivot for each of the node's objects. In this way, for each node N we get a list of triplets $\langle O_i, NN(O_i, N), \delta(O_i, NN(O_i, N))\rangle$, which we call the *nearest-neighbor graph* (NN-graph) of node N, see Figure 2a.

The result is *M*-tree*, an extension of M-tree, where the nodes are additionally equipped by NN-graphs, i.e. a routing entry is defined as

$$rout_l(O_i) = [O_i, r_{O_i}, \delta(O_i, Par(O_i)), \langle NN(O_i), \delta(O_i, NN(O_i))\rangle, ptr(T(O_i))]$$

while the ground entry is defined as

$$grnd(O_i) = [O_i, \delta(O_i, Par(O_i)), \langle NN(O_i), \delta(O_i, NN(O_i))\rangle]$$

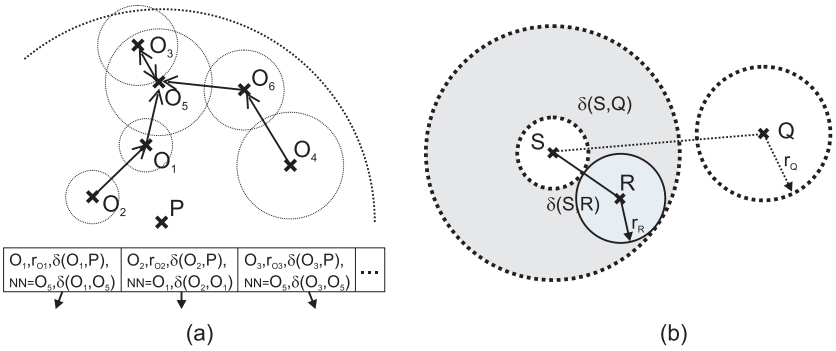

Fig. 2. (a) NN-graph in M*-tree node (b) Filtering using sacrifice pivot

4.2 Query processing

In addition to M-tree's basic and parent filtering, in M*-tree we can utilize the NN-graph when filtering non-relevant routing/ground entries from the search.

NN-graph filtering. The filtering using NN-graph is similar to the parent filtering, however, instead of the parent, we use an object S from the node. First, we have to select such object S; then its distance to the query object Q is explicitly computed. We call the object S a *sacrifice pivot* (or simply sacrifice), since to "rescue" other objects from basic filtering, this one must be "sacrificed" (i.e. the distance to the query has to be computed).

Lemma 1 *(NN-graph filtering)*

Let entry(R) be a routing or ground entry in node N to be checked against a range query (Q, r_Q). Let entry(S) be a sacrifice entry in node N, which is R's nearest neighbor, or R is nearest neighbor to S. Then if

$$|\delta(S,Q) - \delta(S,R)| > r_R + r_Q$$

the entry(R) does not overlap the query and we can safely exclude entry(R) from further processing (for a ground entry $r_R = 0$).

Proof: Follows immediately from the triangle inequality. (a) Since $\delta(S,R)+r_R$ is the upper-bound distance of any object in (R, r_R) to S, we can extend or reduce the radius of rout(S) to $\delta(S,R)+r_R$ and then perform the standard overlap check between (Q, r_Q) and $(S, \delta(S,R)+r_R)$, i.e. $\delta(S,Q)-\delta(S,R) > r_R+r_Q$. (b) Since $\delta(S,R) - r_R$ is the lower-bound distance of any object in (R, r_R) to S, we can check whether the query (Q, r_Q) lies entirely inside the hole $(S, \delta(S,R) - r_R)$, i.e. $\delta(S,R) - \delta(S,Q) > r_R + r_Q$. ■

When using a sacrifice S, all objects which are *reverse nearest neighbors* (RNNs) to S can be passed to the NN-graph filtering (in addition to the single nearest neighbor of S). The reverse nearest neighbors are objects in N having S as their nearest neighbor. Note that for different sacrifices S_i within the same node N, their sets of RNNs are disjoint. The operator RNN can be defined as $RNN : \mathbb{S} \times Nodes \mapsto 2^{\mathbb{S}\times\mathbb{R}^+}$, where for a given sacrifice S and a node N, the operator RNN returns a set of pairs $\langle O_i, d_i \rangle$ of reverse nearest neighbors.

The objects returned by RNN operator can be retrieved from the NN-graph without a need of distance computation (i.e. "for free"). See the NN-graph filtering pseudocode in Listing 1.

Listing 1 *(NN-graph filtering)*

```
set FilterByNNGraph(Node N, sacrifice ⟨S, δ(S,Q)⟩, RQuery (Q, r_Q)) {
    set notFiltered = ∅
    for each entry(O_j) in RNN(S, N) do
        if |δ(S,Q) − δ(S,O_j)| > r_Q + r_Oj then
            filtered[entry(O_j)] = true;
        else
            add entry(O_j) to notFiltered
    return notFiltered
}
```

Range query. When implementing a query processing, the tree structure is traversed such that non-overlapping nodes (their parent regions do not overlap the query ball) are excluded from further processing (filtered out). In addition to the basic and parent filtering, in M*-tree we can use the NN-graph filtering step (inserted after the step of parent filtering and before the basic filtering), while we hope some distance computations needed by basic filtering after an unsuccessful parent filtering will be saved.

When processing a node N, there arises a question how to choose the ordering of N's entries passed to NN-graph filtering as sacrifices. We will discuss various orderings in Section 4.2, however, now suppose we already have a heuristic function $\mathcal{H}(N)$ which returns an ordering on entries in N. In this order the potential sacrifices are initially inserted into a queue SQ.

In each step of a node processing, the first object S_i from SQ (a sacrifice candidate) is fetched. Then, the sacrifice candidate is checked whether it can be filtered by parent filtering. If not, the sacrifice candidate becomes a regular sacrifice, so distance from Q to S_i is computed, while all S_i's RNNs (objects in $RNN(S_i, N)$) are tried to be filtered out by NN-graph filtering. Note the non-filtered RNNs (remaining also in SQ) surely become sacrifices, because RNN sets are disjoint, i.e., an object is passed at most once to the NN-graph filtering. Hence, we can immediately move the non-filtered RNNs to the beginning of SQ – this prevents some "NN-filterable" objects in SQ to become sacrifices. See the range query algorithm in Listing 2.

Listing 2 *(range query algorithm)*

```
QueryResult RangeQuery(Node N, RQuery (Q, r_Q), ordering heuristic H) {
    let P be the parent routing object of N
            /* if N is root then δ(O_i, P)=δ(P, Q)=0 */
    let filtered be an array of boolean flags, size of filtered is |N|
    set filtered[entry(O_i)]=false, ∀entry(O_i) ∈ N
    let SQ be a queue filled with all entries of N, ordered by H(N)

    if N is not a leaf then {
        /* parent filtering */
        for each rout(O_i) in N do
            if |δ(P, Q) − δ(O_i, P)| > r_Q + r_{O_i} then
                filtered[rout(O_i)] = true;
        /* NN-graph filtering */
        while SQ not empty
            fetch rout(S_i) from the beginning of SQ
            if not filtered[rout(S_i)] then
                compute δ(S_i, Q)
                NF = FilterByNNGraph(N, ⟨S_i, δ(S_i, Q)⟩, (Q, r_Q))
                move all entries in SQ ∩ NF to the beginning of SQ
                /* basic filtering */
                if δ(S_i, Q) ≤ r_Q + r_{S_i} then
                    RangeQuery(ptr(T(S_i)), (Q, r_Q), H)
    } else {
        /* parent filtering */
        for each grnd(O_i) in N do
            if |δ(P, Q) − δ(O_i, P)| > r_Q then
                filtered[grnd(O_i)] = true;
        /* NN-graph filtering */
        while SQ not empty
            fetch grnd(S_i) from the beginning of SQ
            if not filtered[grnd(S_i)] then
                compute δ(S_i, Q)
                NF = FilterByNNGraph(N, ⟨S_i, δ(S_i, Q)⟩, (Q, r_Q))
                move all entries in SQ ∩ NF to the beginning of SQ
                /* basic filtering */
                if δ(S_i, Q) ≤ r_Q then
                    add S_i to the query result
    }
}
```

kNN query. The kNN algorithm is a bit more difficult, since the query radius r_Q is not known at the beginning of kNN search. In [7] the authors applied a modification of the well-known heuristic (based on priority queue) to the M-tree. Due to lack of space we omit the listing of kNN pseudocode, however, its form can be easily derived from the original M-tree's kNN algorithm and the M*-tree's range query implementation presented above (for the code see [11]).

Choosing the Sacrifices. The order in which individual entries are treated as sacrifices is crucial for the algorithms' efficiency. Virtually, all entries of a node can serve as sacrifices, however, when a good sacrifice is chosen at the beginning, many of others can be filtered out, so the node processing requires less sacrifices (distance computations, actually). We propose several heuristic functions $\mathcal{H}$ which order all the entries in a node, thus setting a priority in which individual entries should serve as sacrifices when processing a query.

- **hMaxRNNCount**
  An ordering based on the number of RNNs belonging to an entry e_i,
  i.e. $|RNN(N, e_i)|$.
  Hypothesis: The more RNNs, the greater probability the sacrifice would filter more entries.
- **hMinRNNDistance**
  An ordering based on the entry's closest NN or RNN,
  i.e. $min\{\delta(e_i, O_i)\}, \forall O_i \in RNN(N, e_i) \cup NN(N, e_i)$.
  Hypothesis: An entry close to (R)NN stands for a close pivot, so there is a greater probability of effective filtering (following the closeness criterion).
- **hMinToParentDistance**
  An ordering based on the entry's to-parent distance.
  Hypothesis: The lower to-parent distance, the greater probability that the entry is close to the query; for such an entry the basic filtering is unavoidable, so we can use it as a sacrifice sooner.

4.3 M*-tree Construction

The M*-tree construction consists of inserting new data objects into leaves, and of splitting overflowed nodes (see Listing 3). When splitting, the NN-graphs of the new nodes must be created from scratch. However, this does not imply additional computation costs, since we use M-tree's `MinMaxRad` splitting policy (originally denoted `mM_UB_RAD`) which computes all the pairwise distances among entries of the node being split (the `MinMaxRad` has been proved to perform the best [7]). Thus, these distances are reused when building the NN-graphs.

The search for the target leaf can use NN-graph filtering as well (see Listing 4). In the first phase, a candidate node is tried to be found, which spatially covers the inserted object. If more such candidates are found, the one with shortest distance to the inserted object is chosen. In case no candidate was found in the first phase, in the second phase the closest routing entry is determined. The search for candidate nodes recursively continues until a leaf is reached.

Listing 3 *(dynamic object insertion)*

```
Insert(Object O_i) {
    let N be the root node
    node = FindLeaf(N,O_i)
    store ground entry grnd(O_i) in the the target leaf node
    update radii in the insertion path
    while node is overflowed then
        /* split the node, create NN-graphs, produce two routing entries, and return the parent node */
        node = Split(node)
        insert (update) the two new routing entries into node
}
```

When inserting a new ground entry into a leaf (or a routing entry into an inner node after splitting), the existing NN-graph has to be updated. Although searching for the nearest neighbor of the new entry could utilize the NN-graph filtering (in a similar way as in kNN query), a check whether the new entry became the new nearest neighbor to some of the old entries would lead to computation of all the distances needed for a NN-graph update. Thus, we consider simple update of the NN-graph, where all the distances between the new entry and the old entries are computed.

Listing 4 *(navigating to the leaf for insertion)*

```
Node FindLeaf(Node N, Object New) {
    if N is a leaf node then
        return N
    let P be the parent routing object of N          /* if N is root then δ(O_i, P)=δ(P, New)=0 */
    let filtered be an array of boolean flags, size of filtered is |N|
    set filtered[entry(O_i)]=false, ∀entry(O_i) ∈ N
    let usedSacrifices = ∅ be an empty set
    let SQ be a queue filled with all entries of N, ordered by H(N)
        set candidateEntry = null
        minDist = ∞
    while SQ not empty {
        fetch rout(S_i) from the beginning of SQ
        /* parent filtering */
        if |δ(P, New) − δ(S_i, P)| > minDist + r_{S_i} then
            filtered[rout(S_i)] = true;
        if not filtered[rout(S_i)] then {
            compute δ(S_i, New)
            insert ⟨S_i, δ(S_i, New)⟩ into usedSacrifices
            /* NN-graph filtering */
            NF = ∅
            for each S_j in usedSacrifices do
                NF = NF ∪ FilterByNNGraph(N, ⟨S_j, δ(S_j, New)⟩, (New,minDist))
            move all entries in SQ ∩ NF to the beginning of SQ
            if δ(S_i, New) ≤ r_{S_i} and δ(S_i, New) ≤ minDist then
                candidateEntry = S_i
                minDist = δ(S_i, New)
        }
    }
    if candidateEntry = null then {
        do the same as in the previous while cycle, the difference is just in the condition when updating
        a candidate entry, which is relaxed to:
            if δ(S_i, New) ≤ minDist then
                candidateEntry = S_i
                minDist = δ(S_i, New)
    }
    return FindLeaf(ptr(T(candidateEntry)), New)
}
```

4.4 Analysis of Computation Costs

The worst case time complexities of single object insertion into M*-tree as well as of querying are the same as in the M-tree, i.e. $O(log\ n)$ in case of insertion and $O(n)$ in case of querying, where n is the dataset size. Hence, we are rather interested in typical reduction/increase of absolute computation costs[3] exhibited by M*-tree, with respect to the original M-tree.

M*-tree Construction Costs. When inserting a new object into M*-tree, the navigation to the target leaf makes use of NN-graph filtering, so we achieve faster navigation. However, for insertion into the leaf itself the update of leaf's NN-graph is needed, which takes m distance computations for M*-tree instead of no computation for M-tree (where m is the maximum number of entries in a leaf). On the other side, the expensive splitting of a node does not require any additional distance computation, since all pairwise distances have to be computed to partition the node, regardless of using M-tree or M*-tree.

M*-tree Querying Costs. The range search is always more efficient in M*-tree than in M-tree, because only such entries are chosen as sacrifices which cannot be filtered by the parent, so for them distance computation is unavoidable. On the other side, due to NN-graph filtering some of the entries can be filtered before they become a sacrifice, thus distance computations are reduced in this case.

5 PM*-tree

To show the benefits of NN-graph filtering also on other members of the M-tree family, we have implemented *PM*-tree*, a NN-graph-enhanced extension of the PM-tree (see the next section for a brief overview). The only structural extension over the PM-tree are the NN-graphs in nodes, similarly like M*-tree extends the M-tree.

5.1 PM-tree

The idea of PM-tree [10, 13] is to combine the hierarchy of M-tree with a set of p global pivots. In a PM-tree's routing entry the original M-tree-inherited ball region is further cut off by a set of rings (centered in the global pivots), so the region volume becomes smaller (see Figure 3a). Similarly, the PM-tree ground entries are extended by distances to the pivots (which are also interpreted as rings due to approximations). Each ring stored in a routing/ground entry represents a distance range (bounding the underlying data) with respect to a particular pivot. The combination of all the p entry's ranges produces a p-dimensional minimum bounding rectangle (MBR), hence, the global pivots actually map the

[3] The NN-graph filtering is used just to reduce the computation costs, the I/O costs are the same as in the case of M-tree.

metric regions/data into a "pivot space" of dimensionality p (see Figure 3b). The number of pivots can be defined separately for routing and ground entries – we typically choose less pivots for ground entries to reduce storage costs.

Prior to the standard M-tree "ball filtering" (either basic or parent filtering), a query ball mapped into a hyper-cube in the pivot space is checked for an overlap with routing/ground entry's MBRs – if they do not overlap, the entry is filtered out without a distance computation (otherwise needed in M-tree's basic filtering). Actually, the overlap check can be also understood as L_∞ filtering (i.e. if the L_∞ distance from a region to the query object Q is greater than r_Q, the region is not overlapped by query).

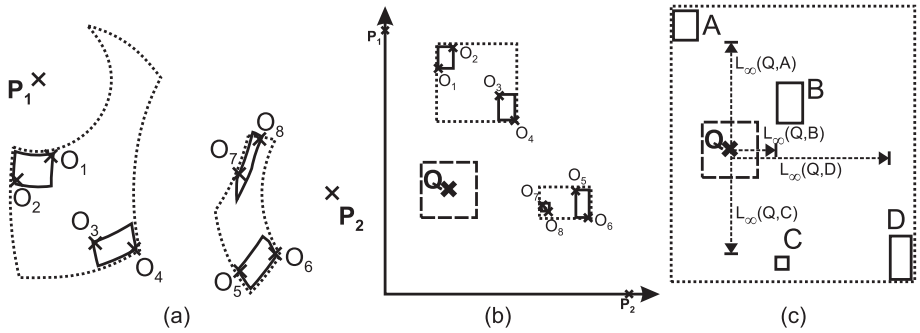

Fig. 3. Hierarchy of PM-tree regions (using global pivots P_1, P_2): (a) metric space view (b) pivot space view. (c) L_∞ distance from Q to MBRs of routing/ground entries.

In PM*-tree we suppose the following ordering of steps when trying to filter a non-relevant node:

parent filtering → L_∞ filtering → **NN-filtering** → basic filtering

5.2 Choosing the Sacrifices

The idea of L_∞ filtering can be also used to propose a PM*-tree-specific heuristics for choosing the sacrifice ordering (in addition to the ones proposed for M*-tree).

- **hMinLmaxDistance**
  An ordering based on the minimum L_∞ distance from Q to the entry's MBR (see Figure 3c).
  Hypothesis: The smaller L_∞ distance, the greater probability that also the δ distance is small, so that the entry has to be filtered by basic filtering (requiring a distance computation).
- **hMaxLmaxDistance**
  An ordering based on the maximum L_∞ distance from Q to the entry's MBR.
  Hypothesis: The greater L_∞ distance, the greater probability that also the δ distance is great, so the entry's RNNs are far from the query and could be filtered by NN-graph.

6 Experimental results

To verify the impact of NN-graph filtering, we have performed extensive experimentation with M*-tree and PM*-tree on three datasets. The sets of query objects were selected as 500 random objects from each dataset, while the query sets were excluded from indexing. We have monitored the computation costs (number of distance computations) required to index the datasets, as well as costs needed to answer range and kNN queries. Each query test unit consisted of 500 query objects and the results were averaged. The computation costs for querying on PM(*)-tree do not include the external distance computations needed to map the query into the pivot space (this overhead is equal to the number of pivots used, and cannot be affected by filtering techniques).

The datasets were indexed for varying dimensionality, capacity of entries per inner node, and the number of pivots (in case of PM(*)-tree). Unless otherwise stated, the PM-tree and PM*-tree indexes were built using 64 pivots (64 in inner nodes and 32 in leaf nodes). Moreover, although the M-tree and PM-tree indexes are designed for database environments, in this paper we are interested just in computation costs (because the NN-graph filtering cannot affect I/O costs). Therefore, rather than fixing a disk page size used for storage of a node, we specify the inner node capacity (maximum of entries stored within an inner node – set to 50 in most experiments).

6.1 Corel dataset

The first set of experiments was performed on the *Corel dataset* [8], consisting of 65,615 feature vectors of images (we have used the color moments). As indexing metric the L_1 distance was used. Unless otherwise stated, we have indexed the first 8 out of 32 dimensions. In Table 1 see the statistics obtained when indexing the Corel dataset – the numbers of distance computations needed to index all the vectors and the index file sizes. The computation costs and index sizes of M*-tree/PM*-tree are represented as percentual growth with respect to the M-tree/PM-tree values.

index type	construction costs	index file size
M-tree	3,708,968	4.7MB
M*-tree	+22%	+25.5%
PM-tree(64,32)	18,522,252	8.8MB
PM*-tree(64,32)	+25.6%	+0%

Table 1. Corel indexing statistics.

In Figure 4a see the M-tree and M*-tree querying performance with respect to increasing capacity of nodes. We can see the M-tree performance improves up to the capacity of 30 entries, while M*-tree steadily improves up to the capacity of 100 entries – here the M*-tree is by up to 45% more efficient than M-tree. The most effective heuristic is the `hMaxRNNCount`.

The impact of increasing PM*-tree node capacities is presented in Figure 4b. The PM*-tree with `hMinLmaxDistance` heuristic performs the best, while the

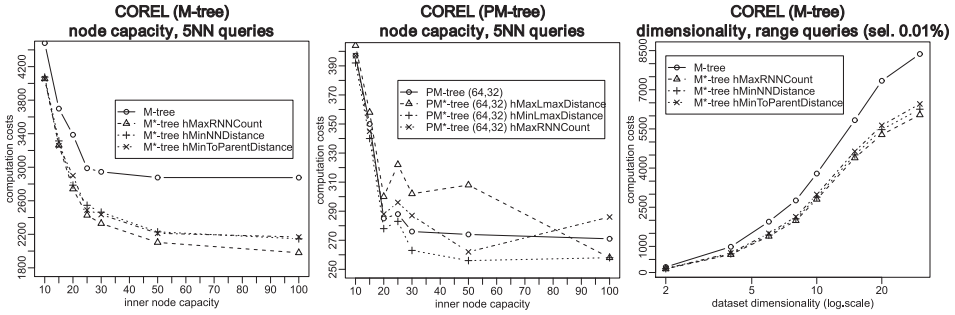

Fig. 4. 5NN queries depending on varying node capacity (a) M*-tree (b) PM*-tree. (c) Range queries depending on increasing dataset dimensionality.

other heuristics are even outperformed by the PM-tree. The gain in efficiency is not so significant as in case of M*-tree (5%).

In Figure 4c see a comparison of M-tree and M*-tree when querying datasets of increasing dimensionality. Again, the `hMaxRNNCount` heuristic performed the best, the efficiency gain of M*-tree was significant – about 40% on average.

6.2 Polygons dataset

The second set of experiments was carried out on the *Polygon* dataset, a synthetic dataset consisting of 1,000,000 randomly generated 2D polygons, each having 5-10 vertices. As the indexing metric the Hausdorff set distance was employed (where L_2 distance was used as partial distance on vertices). In Table 2 see the statistics obtained for the Polygons indexes.

index type	construction costs	index file size
M-tree	70,534,350	148.4MB
M*-tree	+12,1%	+5%
PM-tree(64,32)	291,128,463	202.7MB
PM*-tree(64,32)	+17%	+0%

Table 2. Polygons indexing statistics.

In Figure 5a the M*-tree 1NN performance with respect to the increasing dataset size is presented. To provide a better comparison, the computation costs are represented in proportion of distance computations needed to perform full sequential search on the dataset of given size. The efficiency gain of M*-tree is about 30% on average.

The costs for kNN queries are shown in Figure 5b, we can observe the efficiency gain ranges from 30% in case of 1NN query to 23% in case of 100NN query. As usual, the M*-tree equipped with the `hMaxRNNCount` heuristic performed the best.

The performance of 1NN queries on PM*-tree is presented in Figure 5c, considering increasing number of pivots used. The PM*-tree performance improvement with respect to PM-tree is quite low, ranging from 15% (2 and 4 pivots, heuristic `hMaxLmaxDistance`) to 6% ($\geq$ 8 pivots, heuristic `hMinLmaxDistance`).

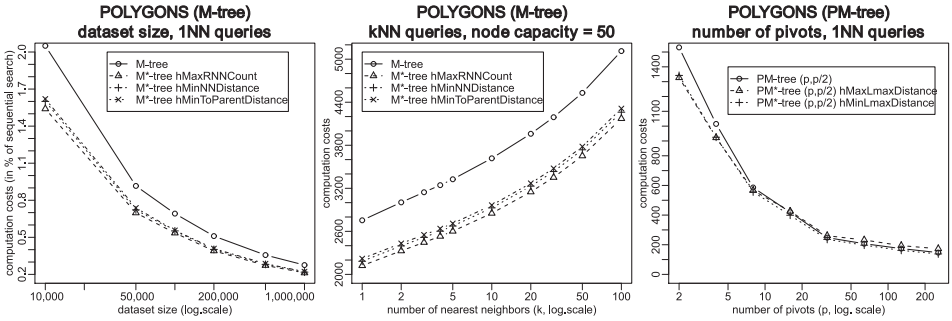

Fig. 5. (a) 1NN queries depending on increasing dataset size. (b) kNN queries (c) 1NN queries depending on increasing number of pivots used in PM-tree.

6.3 GenBank dataset

The last dataset in experiments was created by sampling 250,000 strings of protein sequences (of lengths 50-100) from the *GenBank* file `rel147` [1]. The edit distance was used to index the GenBank dataset. In Table 3 see the indexing statistics obtained for the GenBank dataset.

index type	construction costs	index file size
M-tree	17,726,084	54.5M
M*-tree	+38,9%	+18,2%
PM-tree(64,32)	77,316,482	66.8MB
PM*-tree(64,32)	+20.6%	+0%

Table 3. GenBank indexing statistics.

In Figure 6 see the results for kNN queries on M*-tree and PM*-tree, respectively. Note that the GenBank dataset is generally hard to index, the best achieved results for 1NN by M*-tree and PM*-tree are 136,619 (111,086 respectively) distance computations, i.e. an equivalent of about half of the sequential search. Nevertheless, in these hard conditions the M*-tree and PM*-tree have outperformed the M-tree and PM-tree quite significantly, by up to 20%. As in the previous experiments, the `hMaxRNNCount` heuristic on M*-tree and `hMinLmaxDistance` heuristic on PM*-tree performed the best.

6.4 Summary

The construction costs and index file sizes of all M*-tree and PM*-tree indexes exhibited an increase, ranging from 5% to 25%. For PM*-tree the increase in index file size was negligible in all cases.

The results on the small-sized Corel dataset have shown the M*-tree can significantly outperform the M-tree. However, the PM*-tree performed only slightly better than PM-tree. We suppose the superior performance of PM-tree simply gives only a little room for improvements. Note the dataset of size $\approx$ 65,000 can

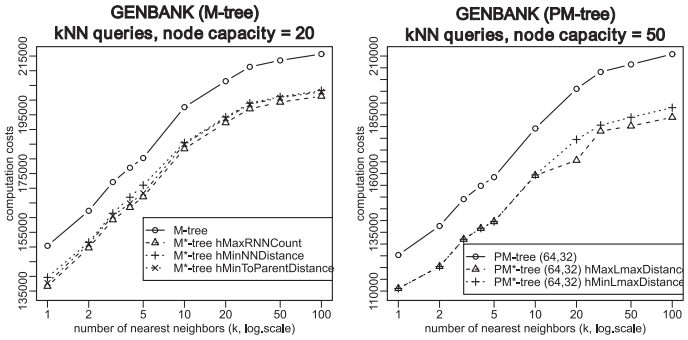

Fig. 6. kNN queries (a) M-tree (b) PM-tree.

be searched by the PM-tree for less than 300 distance computation – this corresponds, for example, to six paths of length 3 in the PM-tree where 15 entries per node must be filtered by basic filtering.

We suppose the extent of M*-tree/PM*-tree performance gain is related to the *intrinsic dimensionality* [4] of the respective dataset. The high-dimensional GenBank dataset is hard to index, so any "help" by additional filtering techniques (like the NN-graph filtering) would result in better pruning of index subtrees. On the other side, when considering the low-dimensional Corel and Polygons datasets, the PM-tree alone is extremely successful (up to 10x faster than M-tree), so we cannot achieve a significant gain in performance.

7 Conclusions

We have proposed an extension of M-tree family, based on utilization of nearest-neighbor graphs in tree nodes. We have shown the NN-graphs can be successfully implemented into index structures designed for a kind of local-pivot filtering. The improvement is based on using so-called "sacrifices" – selected entries in the tree node which serve as local pivots to all entries being reverse nearest neighbors (RNNs) to a sacrifice. Since the distances from a sacrifice to its RNNs are precomputed in the NN-graph, we could prune several subtrees for just one distance computation. We have proposed several heuristics on choosing the sacrifices, and the modified insertion, range and kNN query algorithms.

7.1 Future Work

The properties of NN-graph filtering open possibilities for other applications. Generally, the metric access methods based on compact partitioning using local pivots could be extended by NN-graphs. In the future we would like to integrate the NN-graph filtering into other metric access methods, as a supplement to their own filtering techniques.

Acknowledgments

This research has been supported in part by grants GAČR 201/05/P036 provided by the Czech Science Foundation, and "Information Society program" grant number 1ET100300419.

References

1. D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp, and D. L. Wheeler. Genbank. *Nucleic Acids Res*, 28(1):15–18, January 2000.
2. B. Bustos, G. Navarro, and E. Chávez. Pivot selection techniques for proximity searching in metric spaces. *Pattern Recognition Letters*, 24(14):2357–2366, 2003.
3. B. Bustos and T. Skopal. Dynamic Similarity Search in Multi-Metric Spaces. In *Proceedings of ACM Multimedia, MIR workshop*, pages 137–146. ACM Press, 2006.
4. E. Chávez and G. Navarro. A Probabilistic Spell for the Curse of Dimensionality. In *ALENEX'01, LNCS 2153*, pages 147–160. Springer, 2001.
5. P. Ciaccia and M. Patella. The M^2-tree: Processing Complex Multi-Feature Queries with Just One Index. In *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*, Zurich, Switzerland, June 2000.
6. P. Ciaccia and M. Patella. Searching in metric spaces with user-defined and approximate distances. *ACM Database Systems*, 27(4):398–437, 2002.
7. P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *VLDB'97*, pages 426–435, 1997.
8. S. Hettich and S. Bay. The UCI KDD archive [http://kdd.ics.uci.edu], 1999.
9. M. L. Micó, J. Oncina, and E. Vidal. An algorithm for finding nearest neighbour in constant average time with a linear space complexity. In *Int. Cnf. on Pattern Recog.*, 1992.
10. T. Skopal. Pivoting M-tree: A Metric Access Method for Efficient Similarity Search. In *Proceedings of the 4th annual workshop DATESO, Desná, Czech Republic, ISBN 80-248-0457-3, also available at CEUR, Volume 98, ISSN 1613-0073,* `http://www.ceur-ws.org/Vol-98`, pages 21–31, 2004.
11. T. Skopal and D. Hoksza. Electronic supplement for this paper, `http://urtax.ms.mff.cuni.cz/skopal/pub/suppADBIS07.pdf`, 2007.
12. T. Skopal, J. Pokorný, M. Krátký, and V. Snášel. Revisiting M-tree Building Principles. In *ADBIS, Dresden*, pages 148–162. LNCS 2798, Springer, 2003.
13. T. Skopal, J. Pokorný, and V. Snášel. Nearest Neighbours Search using the PM-tree. In *DASFAA '05, Beijing, China*, pages 803–815. LNCS 3453, Springer, 2005.
14. C. Traina Jr., A. Traina, B. Seeger, and C. Faloutsos. Slim-Trees: High performance metric trees minimizing overlap between nodes. *Lecture Notes in Computer Science*, 1777, 2000.
15. P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity Search: The Metric Space Approach (Advances in Database Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.
16. X. Zhou, G. Wang, J. Y. Xu, and G. Yu. M^+-tree: A New Dynamical Multidimensional Index for Metric Spaces. In *Proceedings of the Fourteenth Australasian Database Conference - ADC'03, Adelaide, Australia*, 2003.
17. X. Zhou, G. Wang, X. Zhou, and G. Yu. BM+-Tree: A Hyperplane-Based Index Method for High-Dimensional Metric Spaces. In *DASFAA*, pages 398–409, 2005.