On Reinsertions in M-tree

Jakub Lokoč
Charles University in Prague, FMP
Department of Software Engineering,
Malostran. nám. 25, Prague, Czech Rep.
jakub.lokoc@mff.cuni.cz

Tomáš Skopal
Charles University in Prague, FMP
Department of Software Engineering,
Malostran. nám. 25, Prague, Czech Rep.
tomas.skopal@mff.cuni.cz

Abstract

In this paper we introduce a new M-tree building method, utilizing the classic idea of forced reinsertions. In case a leaf is about to split, some distant objects are removed from the leaf (reducing the covering radius), and then again inserted into the M-tree in a usual way. A regular leaf split is performed only after a series of unsuccessful reinsertion attempts. We expect the forced reinsertions will result in more compact M-tree hierarchies (i.e., more efficient query processing), while the index construction costs should be kept as low as possible. Considering both low construction costs and low querying costs, we examine several combinations of construction policies with reinsertions. The experiments show that forced reinsertions could significantly decrease the number of distance computations, thus speeding up indexing as well as querying.

1 Introduction

During the last decade, the metric access methods (MAMs) [15] have becoming a respected tool for efficient similarity search in multimedia databases, time series, biometric databases and many other collections of unstructured data entities. A task common to all MAMs is to quickly retrieve database objects relevant to a similarity query (either a range query or a kNN query), where the (dis)similarity between two objects O_i, O_j is modeled by a metric distance $\delta(O_i, O_j)$. Since the evaluation of a single distance value $\delta(\cdot,\cdot)$ is considered as expensive, the MAMs are designed to minimize the number of distance computations spent during query processing (at the cost of more or less expensive indexing). Due to the metric properties of $\delta(\cdot,\cdot)$, MAMs organize the database objects within regions (stored in an index structure) which are subsequently used to cheaply filter out non-relevant (sets of) database objects when querying. So far, many MAMs have been developed, e.g., (m)vp-tree, slim-tree, M-tree, PM-Tree, D-index and many others [15, 8].

In this paper we focus on dynamic construction of M-tree utilizing the technique of forced reinsertions. The experiments have shown the forced reinsertions could not only lead to faster querying, but also to better querying/construction trade-off when compared to other M-tree construction methods.

2 M-tree

The M-tree [5] is a dynamic, balanced and persistent data structure suitable for indexing of large metric databases. The structure of M-tree represents a hierarchy of nested ball regions, where data is stored in leaves (see Figure 1). Every node has a capacity of m entries and a minimal occupation m_{min} (only the root node is allowed to be underflowed below m_{min}). The inner nodes consist of routing entries $rout(O_r)$:

$$rout(O_r) = [O_r, ptr(T(O_r)), r_{O_r}, \delta(O_r, P(O_r))]$$

Where O_r is a routing object, $ptr(T(O_r))$ is a pointer to a subtree $T(O_r)$, r_{O_r} is a covering radius ($\forall O_i \in T(O_r), r_{O_r} \geq \delta(O_r, O_i)$) and the last one is a distance to the parent object $P(O_r)$ denoted as $\delta(O_r, P(O_r))$. The parent distance is not defined for entries in the root. A leaf (ground) entry has a format:

$$grnd(O_i) = [O_i, oid(O_i), \delta(O_i, P(O_i))],$$

where O_i and $\delta(O_i, P(O_i))$ are the same as in the routing entry, $oid(O_i)$ is an identifier of the original object (O_i is just a feature object).

Routing entry represents a ball region in the metric space with its center in pivot O_r and radius equal to r_{O_r}. The regions represented by routing entries on the same level may overlap; this has a negative impact on the number of distance computations when a similarity query is processed.

0-7695-3101-6/08 $25.00 © 2008 IEEE
DOI 10.1109/SISAP.2008.10

During last decade, many methods have been developed to challenge the problem of overlaps and of compact M-tree hierarchies; we overview some recent approaches of M-tree enhancements in Section 3 and present our contributing method in Section 4.

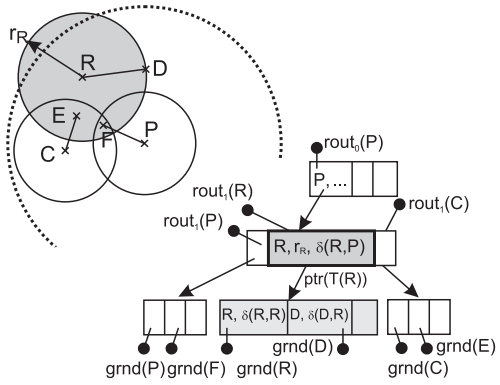

Figure 1. An M-tree hierarchy.

2.1 Building the M-tree

An M-tree is built in the bottom-up fashion (like B-tree, R-tree), that is, data objects are inserted into the leaf nodes. When a leaf is overfull, a split is performed – a new leaf is created and some objects are moved from the original leaf into the new one. Two new routing entries are created (one for the original updated leaf and one for the new leaf) and inserted into the parent node (entry for the original leaf is just replaced). All distances between ground entries and the new routing objects are updated. Because of inserting new routing entries the parent (inner) node could be overfull as well – in such case a split is performed in a similar way, recursively. If the root node is split, the M-tree grows by one level.

When building an M-tree, two main problems have to be solved – the leaf selection and node splitting:

2.1.1 Leaf selection

In the original M-tree, a process similar to a point query is performed, in order to find an appropriate leaf for object placement. However, in contrast to a point query, only one vertical path (branch) of the tree is passed. This approach is also referred to as the *single-way* (deterministic) insertion. When navigating the tree, the next node in the path is chosen such that the inserted object fits the appropriate region best (for details we refer to [5, 12]).

2.1.2 Node splitting

The node splitting policy is a significant factor of the M-tree building process. When a node is split, two new routing entries (representing new ball regions) have to be created. To guarantee a compact M-tree hierarchy, the splitting process must ensure the new regions are separated as much as possible, they overlap as least as possible, and they are of minimum volumes (radii).

To best fit these requirements, all the objects in the node are candidates to the routing objects. For each pair of candidate routing objects, the resulting nodes are temporarily created and radius of the greater region is determined. Such pair of candidate routing objects is finally chosen, which has the smallest radius of the greater region (so-called *mM_Rad* choice). This *CLASSIC* approach bears complexity of $O(m^2)$ (where m is capacity of the node). To avoid the quadratic complexity, there were alternative heuristics developed:

- The *RANDOM* approach directly selects two new routing objects at random, which cuts the complexity down to $O(m)$.
- Instead of considering all objects in the node as candidate routing objects, the *SAMPLING* approach selects randomly just s candidates ($s < m$). Then complexity of node splitting is $O(ms)$.

In Figure 2 see the result of leaf splitting using the *CLASSIC* and *SAMPLING* approach.

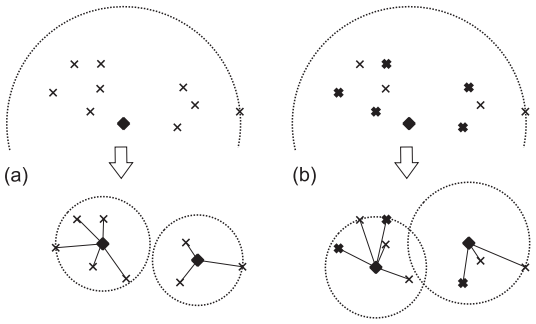

Figure 2. Leaf split using (a) *CLASSIC* **and (b)** *SAMPLING* **approach.**

3 Related work

There have been many variants of M-tree developed. PM-tree [10, 13] which combines the M-tree with pivot-based techniques, M^+-tree [16] which exploits further partitioning of the node by a hyper-plane (i.e., an approach limited to Euclidean vector spaces), M^2-tree [4],

M^3-tree [2] which exploit an aggregation of multiple metrics, very recently introduced M*-tree [11], where each node is additionally equipped by a NN-graph, and many others. In the rest of the paper (and related work) we consider the original structural properties of M-tree [5] (i.e., modified algorithms, not the structure).

The effectiveness of query processing in M-tree heavily depends on the construction algorithm (indexing) used. To increase the search performance, the construction would be expensive, and vice versa. For example, if we use the *RANDOM* node split heuristic, we obtain low construction costs, but the query costs will rapidly increase. In general, to achieve cheap querying, the M-tree's hierarchy of nested ball regions should be as compact as possible → a compact hierarchy means smaller overlaps and smaller volumes → which means less frequent overlaps of non-relevant regions by a query.

Since the structural properties of M-tree are very loose, there are many M-tree hierarchies possible for a single database. Even if we use single construction method, the resulting M-tree hierarchy will still heavily depend on the order in which data objects are inserted (in case of dynamic insertions). An optimal M-tree hierarchy(ies) surely exist(s), however, such construction would require static indexing, and, above all, an exponential construction time. Hence, we would rather prefer an efficient sub-optimal (dynamic) construction, yet producing sufficiently compact hierarchies. In the following we present three such sub-optimal approaches.

3.1 Slim-down algorithm

The main feature of Slim-tree [14] is the *slim-down algorithm*. For each object in each leaf node the algorithm issues a point query to locate more suitable leaves. If there are such leaves the object is moved from the original leaf to the most suitable leaf found (the one having the nearest routing object). The radius in the original node's parent routing entry is decreased. Although the construction costs are high ($O(nlogn)$–$O(n^2)$ where n is database size), the algorithm produces very compact M-tree hierarchies – the query performance could be improved by an order of magnitude. However, slim-down algorithm is a post-processing method (yet not static), it is not very suitable for a dynamically growing database.

In [12], the slim-down algorithm has been generalized for the whole tree. After the leaf level processing, the upper M-tree levels are subsequently processed until the root is reached. The construction costs are comparable to the slim-down algorithm, but the generalized version provides a better query performance.

3.2 Multi-way leaf selection

In the original M-tree the single-way leaf selection is performed. From the global point of view, only one branch is selected on the basis of local conditions. In [12] the authors present *multi-way* leaf selection, which performs a point query to discover all candidate leaf nodes which spatially cover the new object. From this candidates the most suitable leaf (close and non-full) is selected. When compared to the slim-down algorithm the construction is cheaper but the retrieval performance is worse. On the other hand, querying used on multi-way-built index outperforms the single-way version (at the cost of more expensive construction).

3.3 Bulk loading

The basic idea of bulk loading is to create the index from scratch but knowing beforehand the database, thus some optimizations may be performed to obtain a "good" index for that database. Usually, the proposed bulk loading techniques are designed for specific index structures, but there have been proposals for more general algorithms. For example, in [6] the authors propose two generic algorithms for bulk loading, which were tested with different index structures like the R-tree and the Slim-tree. Note that the efficiency of the index may degrade if new objects are inserted after its construction. Specific bulk loading techniques for M-tree were introduced in [3, 9].

4 Forced Reinserting in M-tree

The *forced reinserting* is a well-known technique from R*-tree [1]. The idea is based on easy principle, where some objects are removed from a leaf to avoid a split operation and then inserted in a common way under a hope that the reinserted objects will arrive into more "suitable" leaf(s). There are two basic motivations to consider forced reinsertion as beneficial (considering any B-tree-based spatial/metric index structure). The straightforward (but also weaker) motivation is better node occupancy, that is, forced reinsertions lead to fuller nodes. Second, due to unavoidable node splitting over the time, the compactness of spatial/metric region hierarchy deteriorates (the region volumes and overlaps grow because of spatial aggregations mixing old and new objects/regions). Here the forced reinsertions could serve as an opportunity to move some "bad" (volume- or overlap-inflating) objects from the leaf.

In M-tree, we have to face some specific issues, when implementing forced reinsertions. Basically, when a

new object is inserted into a leaf that is now about to split, some suitable objects from the leaf must be selected and reinserted. The crucial goal is to propose a method aiming to decrease the covering radius of the reinserted leaf as much as possible while aiming to grow the radii of leaves accepting the reinserted objects as little as possible. Here we have to take also the induced leaf splits/reinsertions into account, that is, a forced reinsertion attempt could raise a chain of reinsertions (terminated by regular splits "after a while").

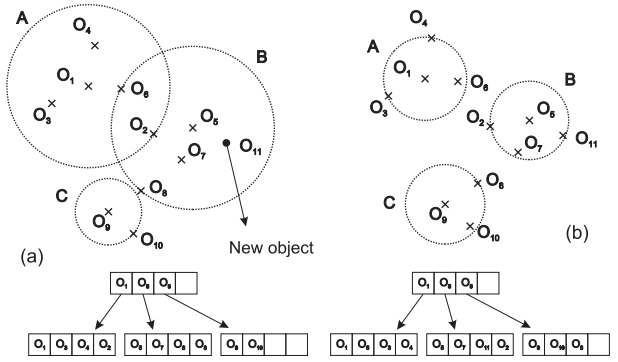

Figure 3. (a) Situation before split. (b) Decreased overlap after reinserting.

As a fundamental assumption, we expect objects located close to the region's "border" have higher probability to be suitably reinserted than the more "centered" ones. Since in an M-tree node the entries are ordered according to their distances to the parent routing entry (region's center), we can select the furthest ones (close to the border) easily.[1] In Figure 3 see a motivation – situation just before a leaf split, and how the split is avoided after a series of (induced) reinsertions. We can see that not only the split was prevented, but the M-tree compactness was improved, too.

In the following we present some details of our approach to forced reinsertions in M-tree.

As mentioned before, we assume the most suitable entries for reinserting are the furthest ones from the parent routing entry. To avoid a leaf split, k furthest entries are removed from the leaf and stored in a temporary main memory stack $\mathcal{S}$. At the same moment the covering radius is updated to the distance to the new furthest entry in the leaf (and so the covering radii of all ancestors). Then, the current entry on the top of $\mathcal{S}$ is reinserted (possibly causing further reinsertion attempts, i.e., filling the stack). The reinsertions are repeated until the stack becomes empty.

[1]Remember the precomputed distances to the routing entry (pivot) are stored in all entries (except entries in the root node).

4.1 Recursion depth

Since a single reinsertion attempt could generally raise a long chain of subsequent reinsertions (the stack is inflating instead of emptying), we would like to limit the number of forced reinsertion attempts to keep the construction costs reasonable. The limit is denoted as a user-defined *recursion depth* parameter, so when the limit of reinsertion attempts is reached, the remaining entries in the stack are reinserted such that only regular splits are allowed from now on (i.e., the stack does not grow anymore).

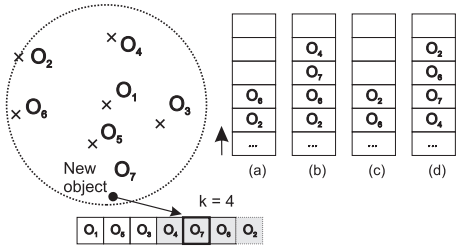

Figure 4. Entries removing strategies: (a) Pessimistic, (b) Optimistic (c) Rev_Pessimistic, (d) Rev_Optimistic.

4.2 Entries removing

We define four strategies (depicted in Figure 4) for entries removing. Let grnd(O_{new},...) be the new inserted entry that invoked the split, and let $\mathbb{E}$ denote the leaf's portion consisting of the k furthest entries. The two former strategies consider situation when $\mathbb{E}$ contains O_{new}. Here, the objects are processed in the descending order (i.e., starting from the furthest entry).

- The *pessimistic* strategy supposes O_{new} will be reinserted again into the same leaf. To prevent this, the removing process is stopped after O_{new} is reached, thus, O_{new} and closer objects are not removed from $\mathbb{E}$. If O_{new} is the furthest object in the node, the regular split has to be performed.
- The *optimistic* strategy removes all objects from $\mathbb{E}$, thus the radius of the leaf region is minimized, while anticipating all objects from $\mathbb{E}$ (possibly including O_{new}) will be reinserted into other leaves.
- *Rev_pessimistic* and *Rev_optimistic* strategies are similar to the former two. The only difference is in the reverse order of processing, that is, the removing starts with the closest entry, so the furthest one is on the top of the stack $\mathcal{S}$.

To provide clear summary, in Listing 1 see the pseudocode of dynamic insertion algorithm enhanced by forced reinsertions.

Listing 1 (insertion with forced reinsertions)

```
let maxRemoved be maximal number of removed entries (user-defined)
let removingStrategy be the type of entries removing (user-defined)
let recursionDepth be the maximal depth of recursion (user-defined)

method Insert(O_new) {
  find leaf L for O_new
  insert O_new into L

  if L is not overfull then
    return

  let 𝔼 be the portion of L with maxRemoved furthest entries (sorted ASC)

  if removingStrategy is Pessimistic or RevPessimistic then
    exclude grnd(O_new,...) and all closer entries from 𝔼

  if |𝔼| > 0 and recursionDepth > 0 then
    for (j = 0; j < |𝔼|; j++)              // remove furthest entries from leaf
      if removingStrategy is RevOptimistic or RevPessimistic then
        𝒮.Push(𝔼.GetEntry(1))
        𝔼.DeleteEntry(1)
      else
        𝒮.Push(𝔼.GetEntry(|𝔼|))
        𝔼.DeleteEntry(|𝔼|)

    decrease radius of L (and possibly of its ancestors)

    while (𝒮 is not empty)                  // reinsert removed entries
      recursionDepth = recursionDepth − 1
      Insert(𝒮.Pop())
  else
    perform regular split of L (and possibly of its ancestors)
}
```

4.3 Construction vs. query efficiency

The benefits of forced reinsertions are two-fold. First, reinsertions could clearly improve the compactness of M-tree (thus the retrieval performance) at the cost of more expensive construction. Furthermore, the second benefit considers the trade-off between indexing and querying performance. We would like to decrease construction costs but simultaneously keep the retrieval costs as low as if used more expensive construction. With forced reinsertions this goal could be carried out. For example, the *CLASSIC* splitting of M-tree node is expensive but brings a faster retrieval, while the *SAMPLING* splitting is cheaper but also leads to slower retrieval. Since the *CLASSIC* splitting could produce M-tree which is compact enough, at some scenarios the employment of forced reinsertions could not bring any further improvement (so only the construction costs grow, but the retrieval performance is not improved). In such case we could rather employ the forced reinsertions together with the *SAMPLING* splitting, in order to achieve retrieval costs similar to that achieved by *CLASSIC* splitting, however, for cheaper construction (somewhere between *SAMPLING* and *CLASSIC* without forced reinsertions). In other words, forced reinsertions could cheaply fix the bad data partitioning caused by *SAMPLING* splitting.

5 Experimental results

We have performed experimentation focusing on the distance computations spent during a query processing and index construction, when (not) using forced reinsertions. We have performed the tests on two different databases, a subset of *Corel* [7] image features (68,040 32-dimensional vectors representing color histograms). As a distance function the Euclidean (L_2) distance has been employed. The second database was a synthetic randomly generated set of 250,000 2D polygons (each polygon consisting of 10–15 vertices). The Hausdorff distance was used to measure similarity of two polygons. The query costs were always averaged for 200 uniformly distributed query objects. We did not perform an inter-MAM comparison; we have focused just on various configurations of M-tree (with or without forced reinsertions). As parameters, we have observed various data dimensionalities, database sizes, node's capacities as well as various forced reinsertion settings.

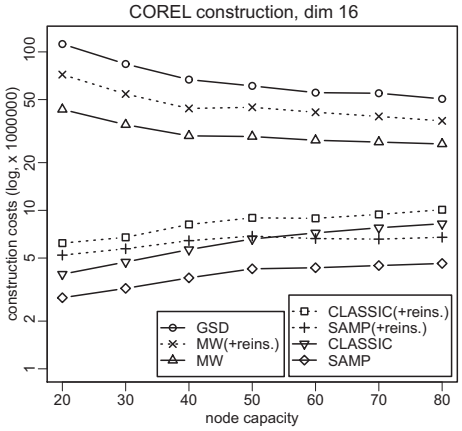

Figure 5. Construction costs for growing node capacities.

5.1 M-tree Construction

We have tested four M-tree building methods: the original (single-way) M-tree insertion with CLASSIC and SAMPLING node splitting serving as a baseline (denoted as CLASSIC and SAMP, respectively), then the generalized slim-down algorithm denoted as GSD,

multi-way leaf selection denoted as MW (GSD and MW used the *CLASSIC* node splitting). For all of them (except GSD) we examined variants with forced reinsertions (label attached by +reins.), so finally 7 ways of M-tree construction were tested. The recursion depth for reinsertions was set to 10 (observed as the best value), the sample size was set to 10% of the node capacity (when using the *SAMPLING* splitting), the maximum number of removed entries per leaf was set to $k = 5$ (observed as the best value, see below). The reverse pessimistic entries removing strategy was used, unless otherwise stated.

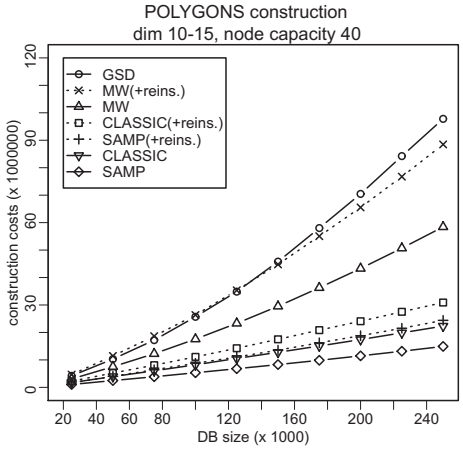

Figure 6. Construction costs for growing database size.

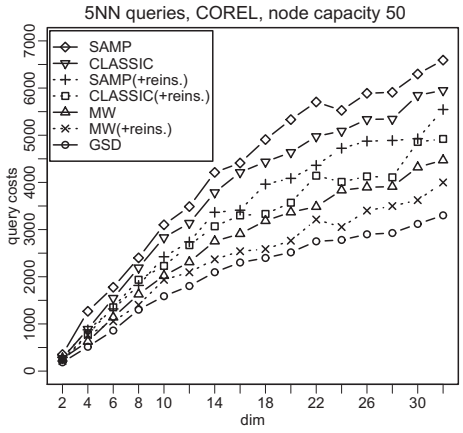

Figure 7. 5NN query costs for varying dimensionality.

In Figure 5 the construction costs for growing node capacities are shown. As we can see, the costs of SAMP grow slowly, which is caused by the subquadratic node splitting complexity. Moreover, for higher node capacities SAMP+reins. becomes cheaper than CLASSIC. As expected, construction costs of GSD are an order of magnitude higher (followed by MW(+reins.)).

The Figure 6 shows that construction costs are linear with respect to the database size (Polygons) for CLASSIC+reins. and SAMP+reins. as well as for their counterparts without reinserting.

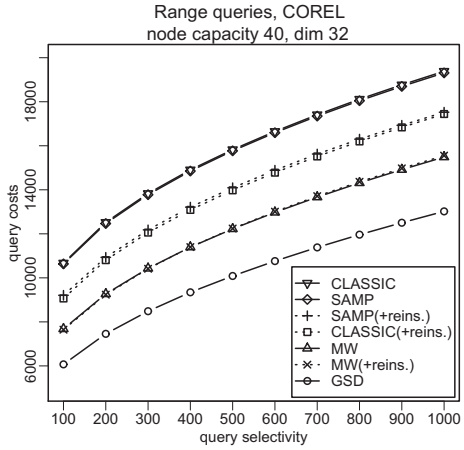

Figure 8. Query costs for growing range query selectivity.

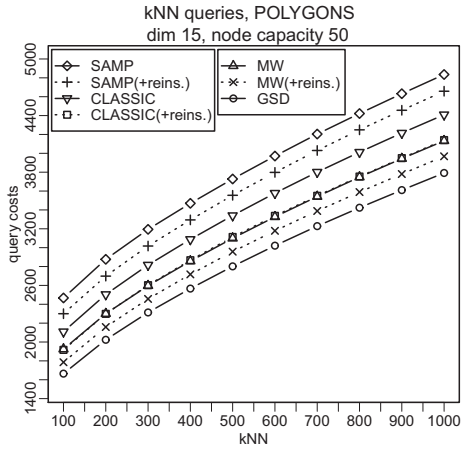

Figure 9. Query costs for kNN queries.

5.2 Range and kNN queries

When considering the Corel database, forced reinserting improves kNN queries significantly. In Figure 7 see the decrease of query costs down to 75% for SAMP+reins. and CLASSIC+reins.. Reinserting combined with the multi-way leaf selection (MW+reins.) decreases query costs down to 80%. Although the construction costs for MW(+reins.) are high, they are still

cheaper than those of the generalized slim-down algorithm.

The next test is presented in Figures 8 and 9, considering growing range query selectivity and the number of the nearest neighbors. It can be observed, that forced reinserting follows the trend shown by the original methods. When looking at Polygons, the CLASSIC+reins. queries are as fast as MW queries, but in contrast to MW the construction costs for CLASSIC+reins. are just 60% of MW construction costs.

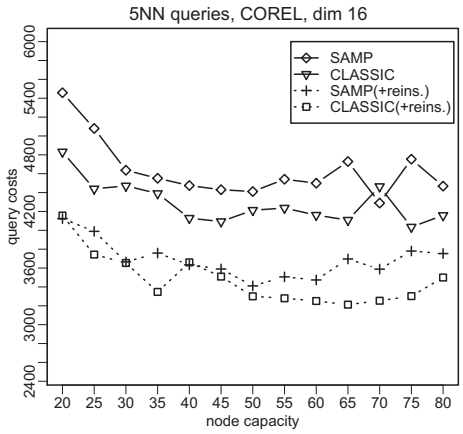

Figure 10. 5NN query costs for a growing node capacity.

Query costs for growing node capacity are presented in the Figure 10. We can see the methods enhanced by forced reinserting outperform the original ones.

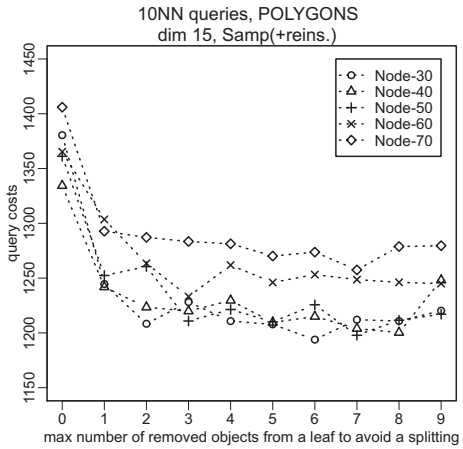

Figure 11. Maximal number of removed entries per leaf.

5.3 Parameters of Forced Reinsertion

In addition to external parameters, we have examined also internal parameters – maximal number of removed entries per leaf and strategies for entries removing (see Section 4.2).

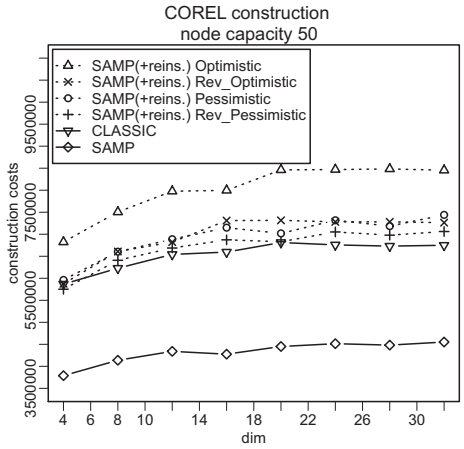

Figure 12. Index construction costs for different removing strategies.

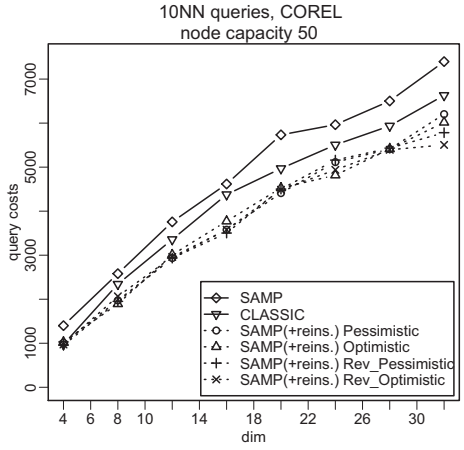

Figure 13. Query costs of indexes with different removing strategies.

The maximal number of removed entries per leaf (see Figure 11) positively affects the query performance just until $k = 5$ is reached. However, for increasing k the construction costs grow, so we fixed $k = 5$ in all the other experiments (as mentioned at the beginning of this section).

In Figures 12 and 13 see the effects of all four strategies for removing entries. The reverse pessimistic strategy won in both low construction costs and good query

performance, hence we have used this strategy in all other experiments (as mentioned at the beginning of this section).

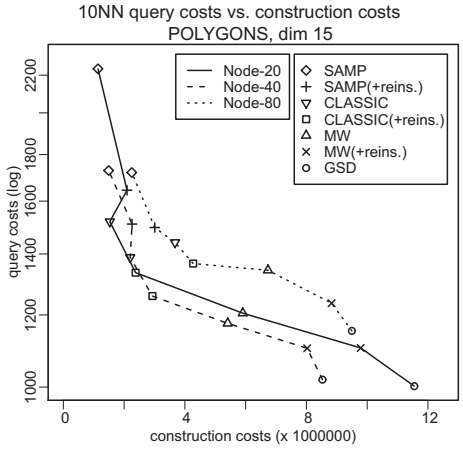

Figure 14. Overall performance of M-tree with or without forced reinsertions.

Finally, the Figure 14 provides an aggregated picture over all the mentioned M-tree construction strategies, considering the construction vs. query performance trade-off. When considering a compromise between construction and query performance, the forced reinsertion improves the CLASSIC and SAMP methods significantly.

6 Conclusion

We have proposed a new M-tree building method, which utilizes the well-known forced reinsertion idea. This approach provides more compact M-tree hierarchies, resulting in better query performance. We have also shown the forced reinsertion can be combined with some cheap node splitting strategies to keep construction costs low, while improving the query costs. Furthermore, forced reinsertion, as a general technique, can be easily combined with other approaches. This feature gives us a flexible tool for tuning the M-tree properties, in order to achieve a desired behavior of the entire indexing/retrieval process.

Acknowledgments

This research has been partially supported by Czech grants: "Information Society program" number 1ET100300419 and Institutional research plan number MSM0021620838.

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, 1990.

[2] 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.

[3] P. Ciaccia and M. Patella. Bulk loading the M-tree, In Proceedings of the 9th Australasian Database Conference (ADC'98), pages 15–26, Perth, Australia, 1998.

[4] P. Ciaccia and M. Patella. The M2-tree: Processing Complex Multi-Feature Queries with Just One Index. In *DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries*, 2000.

[5] 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.

[6] J. V. den Bercken and B. Seeger. An evaluation of generic bulk loading techniques. In *Proc. 27th International Conference on Very Large Data Bases (VLDB'01)*, pages 461–470. Morgan Kaufmann, 2001.

[7] S. Hettich and S. Bay. The UCI KDD archive [http://kdd.ics.uci.edu], 1999.

[8] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.

[9] A. P. Sexton and R. Swinbank. Bulk Loading the M-Tree to Enhance Query Performance. In *BNCOD*, pages 190–202, 2004.

[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. Improving the performance of m-tree family by nearest-neighbor graphs. In *ADBIS*, pages 172–188, 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. 1777:51–65, 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 *ADC*, pages 161–168, 2003.