Optimizing Similarity Search in the M-Tree

Steffen Guhlemann[1] Uwe Petersohn[2] Klaus Meyer-Wegener[3]

Abstract: A topic of growing interest in a wide range of domains is the similarity of data entries. Data sets of genome sequences, text corpora, complex production information, and multimedia content are typically large and unstructured, and it is expensive to compute similarities in them. The only common denominator a data structure for efficient similarity search can rely on are the metric axioms. One such data structure for efficient similarity search in metric spaces is the M-Tree, along with a number of compatible extensions (e.g. Slim-Tree, Bulk Loaded M-Tree, multiway insertion M-Tree, M^2-Tree, etc.). The M-Tree family uses common algorithms for the k-nearest-neighbor and range search. In this paper we present new algorithms for these tasks to considerably improve retrieval performance of all M-Tree-compatible data structures.

Keywords: Metric databases, metric access methods, index structures, multimedia databases, selectivity estimation, similarity search

1 Introduction

Collection and storage of large data sets gives rise to the necessity to also query and process them. The data elements are typically large, unstructured, expensive to process and hardly ever equal to each other. A natural type of query is thus the similarity query.

Hence, there is a need for a general index structure that supports similarity queries on this kind of data. Examples of potential applications are search for genome sequences, fingerprints, and faces [SMZ15], query by example in multimedia databases, machine learning (e.g. k-nearest-neighbor classification or case-based reasoning), etc. Typical query operators are classified in [DD15].

Such an index structure cannot use any structural information, since there is no (common) structure in the data.[4] It can only rely on a generic metric distance function, which must fulfill the axioms of a metric (non negativity, identity of indiscernibles, symmetry, and triangle inequality). Further, this structure has to take into account that a single distance computation can be extremely expensive and that very high dimensional data can fall under the curse of dimensionality. Other requirements are the possibility to store the data on hard disk (i.e. minimize I/O) and to incrementally add or remove entries from the index.

[1] previously TU Dresden, Faculty of Computer Science, Institute for Artificial Intelligence, D-01062 Dresden, Germany, steffenguhlemann@hotmail.com

[2] TU Dresden, Faculty of Computer Science, Institute for Artificial Intelligence, D-01062 Dresden, Germany, Uwe.Petersohn@tu-dresden.de

[3] Friedrich-Alexander-Universität Erlangen-Nürnberg, Faculty of Engineering, Department of Computer Science, Martenstr. 3, D-91058 Erlangen, Germany, klaus.meyer-wegener@fau.de

[4] For some domains like string similarity, special indices using the domain structure exist (e.g. [Fe12] or [Rh10]), but they are not generally applicable.

For similarity search in metric spaces there exists a broad range of structures like BK-Tree [BK73], Fixed Query Tree [Ba94], VP-Tree [Uh91], Bisector Tree [KM83], GNAT [Br95], AESA [Vi86], D-Index [Do03], Metric Index [NBZ11], PPP-Code [NZ14], iDistance [Ja05] and variations of these [CMN01, CMN99, BO97, Yi93, Yi99, DN88, CN00, NVZ92, No93, MOV94, Sh77, BNC03, DGZ03]. A good classification of metric index structures is given by [He09]. Many of these index structures have serious drawbacks, be it for example the restriction to discrete metric spaces (BK-Tree) or a quadratic space complexity (AESA). Further, most of these data structures are inherently static (incremental changes to the stored data are not possible or prohibitively expensive) and only designed to minimize distance computation (and not for example I/O and in-memory tree traversal). An exception is the M-Tree family [CPZ97, Ze06, CP98, Pa99, Sk03, Tr00, Tr02, Ci00] which is thoroughly designed to be a dynamic index structure capable to perform in a broad range of domains and optimizing both I/O and distance computations.[5] This M-Tree family has a compatible structure and shares common query algorithms,[6] which are derived directly from the corresponding queries on a B-Tree with only a few modifications. (Both trees share the basic tree structure.) However, there are subtle differences. For one, the B-Tree divides the space comprehensively and free of overlap, while the M-Tree does not. Second, the B-Tree search only focuses on minimizing the number of disk accesses, while the main goal for the M-Tree is to minimize the number of distance calculations. This leaves room for a more thorough design of the query algorithms with regard to the requirements of similarity search in metric spaces. In this paper new algorithms for the range and the k-nearest-neighbor search are presented. Due to the compatible structure, these new algorithms are applicable to the whole M-Tree family.

The outline of the paper is as follows. In Section 2 the basic structure of the M-Tree family will be presented. Section 3 will introduce some generic optimizations applicable to different kinds of search. The following Sections 4 and 5 will present specific optimizations of the range search and the k-nearest-neighbor search, respectively. Finally Section 6 will show some experimental results.

2 The M-Tree

2.1 The M-Tree Structure

The M-Tree [CPZ97, Ze06] family is used to index similarity in general metric spaces relying only on a metric distance function. The trees grow and shrink dynamically as new data are added or deleted. It is a multi-branch tree that can be configured to minimize I/O. Its basic structure is quite similar to the B-Tree and the R-Tree.

[5] A project seminar [La11] compared different index structures in different domains. The M-Tree had by far the best query performance in terms of necessary distance calculations.

[6] Note, however, that there is a broad range of structures that also pretend to be an extension to the M-Tree, like the Pivoting M-Tree [Sk04], the (B)M^+-Tree [Zh05] or the CM-Tree [AS07], but are not compatible to the M-Tree and also do not have the advantages of the M-Tree. For example the (B)M^+-Tree can only handle Euclidean vector spaces for which better approaches like the kd-Tree exist.

An M-Tree is structured as a hierarchical tree of hypersphere-shaped nodes. Each node entry consists of a routing (pivot) element (which is the center of the hypersphere) and an allowed distance between pivot and data stored below this node (the radius of the hypersphere). Each node can have multiple subnodes up to a predefined capacity limit. All data belonging to a subnode must have a distance to the parent pivot that is smaller than radius of the parent-node hypersphere. Leaf nodes store links to the actual data. The tree grows in a B-Tree-manner bottom-up, i.e., all leafs are at the same level (see Figure 1).

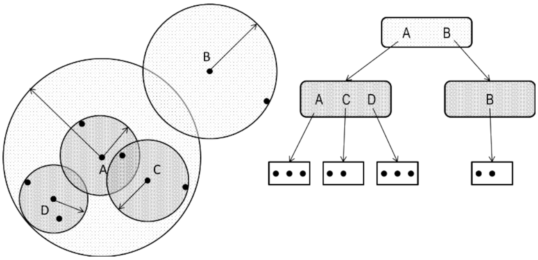

Fig. 1: Structure of an M-Tree [CPZ97]

During insertion of an entry or subnode c into a node n, the distance between n and c must be computed (e.g. to adjust the node radius). As a first optimization, Ciaccia et al. [CPZ97] proposed in their original publication to store the parent-child distances along with the child-node pointers. In search, these precomputed distances are used in conjunction with the triangle inequality to bound the range of the distance of the child pivot to the query object without actually computing it.

2.2 Basic Search Algorithm

Like common tree-search algorithms, similarity search in the M-Tree is basically a hierarchical tree descend pruning nodes whenever possible. In classic main-memory structures the focus is on reducing tree-traversal operations. If data is persisted on paged external memory, a second focus is on reducing I/O operations. For general metric spaces a third optimization goal is necessary: Distance computations have to be avoided as they are usually way more expensive than standard tree-traversal operations, and sometimes even I/O operations.[7]

In general, similarity queries in metric spaces can be rewritten as the search for all data located in a hyperspheric query region centered in a query object. In case of the range query, the radius is fixed as part of the query arguments. In case of a k-nearest-neighbor query, the final search radius has to be determined during search.

The search starts at the root node. It always keeps a queue of unexpanded nodes, which may contain data fulfilling the query. As a node is removed from the queue, its child nodes are

[7] As a simple example consider the Levenshtein edit distance of long texts. A basic distance calculation consumes time and space of $O(N^2)$ where N is the length of the texts. Reading the text from disk will require some time of $O(\mathtt{Seek} + \mathtt{Read} \cdot N)$ where Seek and Read are constants. So, if the text is long enough, computing the distance will easily exceed the time to read the text from disk. The problem is worse for most multimedia domains, where one similarity calculation can be a complex optimization on its own.

examined. If it can be guaranteed that the child node cannot contain search results, it can be pruned. Otherwise it is enqueued in the expansion queue. If a leaf node is removed from the queue, the actual data in it are classified as fulfilling the query criteria or not.

To prune a node n it has to be guaranteed that its hypersphere (center n, radius r_n) does not intersect the query hypersphere (center q, radius r_q).[8] In other words, it must be proven that no data element e below n is closer to q than r_q. In the following this closest possible distance is denoted by $d_n^\perp$ – implicitly referencing the current search center. For an inner node this minimal possible distance is calculated using an (expensive) distance calculation $d_{n,q}$ and the triangle inequality by $d_n^\perp = \max(0, d_{n,q} - r_n)$. On the leaf level, the formula is simplified to $d_e^\perp = d_{e,q}$. A node (or leaf) is pruned, if $d_n^\perp > r_q$.

3 General Search Optimizations

The search approach described in Section 2.2 can be improved in a number of aspects independent of the actual type of search.

3.1 Generalization of Existing Optimizations

During examination of a node n the minimum possible element distance $d_n^\perp$ needs to be calculated to make a justified pruning decision. By default, this calculation involves a distance computation which is the search-cost driver.

A basic idea is to use heuristics to inexpensively retrieve a lower bound $\perp_n$ on the distance $d_{n,q}$. Based on $\perp_n$ a lower bound $d_{n,relaxed}^\perp$ on $d_n^\perp$ can be found without any distance calculation:

$$d_{n,relaxed}^\perp = \max(0, \perp_n - r_n).$$

In certain situations, it is possible to prune n based on $d_{n,relaxed}^\perp$ only, saving the distance calculation $d_{n,q}$ entirely. It has to be observed that $\perp_n$ must never overestimate $d_{n,q}$. Otherwise a node might be pruned which contains valid results, leading to an incorrect search. On the other hand, $\perp_n$ should be as close as possible to $d_{n,q}$ as this allows to prune nodes more often. A further criterion for an efficient search is that the effort of the calculation of $\perp_n$ has to be negligible compared to the computation of $d_{n,q}$. Otherwise only distance computations are saved, but search time is not.

In the literature several examples of such tests exist. Unfortunately they are hard-coded into the respective algorithm without considering generalizability.

[8] Note that in case of a k-nearest-neighbor query the query radius r_q will only be determined during the search. However, there will always be a known upper bound on this radius.

3.1.1 Precomputed Distance to Parent Node

One of the optimizations was proposed by Ciaccia et al. [CPZ97] as part of the classic M-Tree. As described in Section 2.1 each M-Tree node stores the distance to its direct parent node. This neither increases the insert effort[9] nor the storage complexity[10]. Since during search a child node c is only expanded after it has been stated that its parent node p cannot be pruned, the distance $d_{p,q}$ has already been computed at the time of the decision on c. Using the precomputed parent-child distance $d_{c,p}$ stored in c, a quick calculation of a bound for the distance $d_{c,q}$ is possible without actually computing it:

$$d_{c,q} \geq \perp_{n,ParentDist} := |d_{p,q} - d_{c,p}|.$$

3.1.2 AESA Principle

Aronovich and Spiegler [AS07] proposed something as part of the CM-Tree, which can be generalized as node-local AESA [Vi86] principle.[11] Inside each node all bilateral distances between child nodes are stored. If some child node c_i of a parent node is examined (involving the distance computation $d_{c_i,q}$), the distance to other child nodes c_k can be bound using this distance and the precomputed distances d_{c_i,c_k}. The lower bound of the distance to c_k is the maximum of all bounds based on already computed distances to c_i:

$$d_{c_k,q} \geq \perp_{n,AESA} := \max_i \left(d_{c_i,q} - d_{c_i,c_k} \right).$$

Contrary to the basic optimization of [CPZ97] (Section 3.1.1), insert effort and storage complexity (in terms of node capacity) are increased. Further, this tree is not fully compatible to the classic M-Tree, i.e., it needs different insert, delete, and query algorithms.

3.1.3 Domain-specific Heuristics

Bartolini et al. [BCP02] proposed the use of a specific M-Tree structure for indexing Levenshtein edit distances. They developed specific "bag" heuristics to compute a cheaper bound on the edit distance sorting letters into bags.

Their idea can be generalized to allow the M-Tree to make use of domain-specific distance bounds in case such bounds exist. However, as this example shows, the idea must be used with care. The Levenshtein edit distance can be calculated in $O(M \cdot N)$ in terms of the lengths M and N of the two texts. The bag heuristics reduce this time to $O(N + M)$. The

[9] To decide on the parent node to insert a child and to adjust the parent-node radius, this distance must be computed in any case during insert.

[10] Each node stores only one distance. Thus, the storage complexity is $O(1)$. Even in absolute terms this one number is usually negligible as typical metric data (e.g. texts, images, genome sequences) are big. Of course in edge cases (2D-data) this can mean an increase of 33%.

[11] AESA [Vi86] is a similarity index structure that precalculates all bilateral distances. During search these distances can be used to prune objects based on few computed distances and the triangle inequality.

problem here is that search algorithms assume heuristics computations to be nearly for free compared to a single distance computation. That is clearly not the case here – the effects depend on the two string lengths. A better approach would be to use heuristics that can be computed fast in $O(1)$, even if its bounds are not so tight.

3.2 Generalized Use of Bounds

3.2.1 Combination of Bounds

As stated in Section 3.1 multiple domain-specific and domain-independent distance bounds $\perp_n^i$ exist, which have 3 main properties:

- Each $\perp_n^i$ must not overestimate $d_{n,q}$, i.e., $\forall i, n : \perp_n^i \leq d_{n,q}$.
- The $\perp_n^i$ have different precision $Pr_i := d_{n,q} - \perp_n^i$. The smaller Pr_i is, the more distance calculations these heuristics may avoid.
- Each $\perp_n^i$ demands a different computation time T_i. In order to be of any use, T_i must be negligible compared to the time of an actual distance calculation.

Using these properties multiple distance bounds can be combined to a stronger one:

$$\perp_n^{combined} = \max_i \perp_n^i$$

$\perp_n^{combined}$ never overestimates $d_{n,q}$ and has the same or a better precision than any $\perp_n^i$.

This optimization can be applied in a generic manner – i.e., not hard-wiring any specific heuristics into the query algorithm. Domain-specific algorithms can be injected into the index structure either at run time using abstract interface classes describing the properties of a certain metric space, or at compile time using languages like C++ and generic metric-space traits. Each search algorithm would then combine all available heuristics to avoid distance calculations.

Care has to be taken to not replace an expensive distance calculation by a comparably expensive heuristics. As an example consider the optimization described in Section 3.1.3. It has a reduced time complexity of $O(N + M)$ (distance calculation: $O(N \cdot M)$). However, compared to a single in-memory tree traversal or really cheap heuristics (e.g. parent distance - see Section 3.1.1) it is not negligible.[12] Hence, it might be inappropriate to use this optimization for certain data distributions, because it may increase the search time. Further, even if it is used, the algorithm should not just compute the maximum of all bounds, ignoring their different computation times. Instead it should try to subsequently prune a node based on single heuristics (cheap heuristics first). [13]

[12] For the search algorithm it is usually assumed that all tree traversal and heuristics operations are essentially for free compared to a single distance computation. So they are used in abundance. This of course depends on the relative cost of a distance calculation, so it might be valid for typical metric domains (texts, multimedia) but is invalid when for example a low to medium dimensional euclidian distance is used.

[13] [ZS15] propose a cost-benefit ratio to choose between heuristics in the context of multi-feature similarity search.

3.2.2 Upper Bounds

Beside lower bounds on the distance also upper bounds can be computed efficiently and used during search. For example, in case of the parent distance (Section 3.1.1) such upper bound can be computed as

$$d_{n,q} \leq \top_{n,ParentDist} := d_{p,q} + d_{n,p}.$$

Based on this bound, the maximum possible distance $d_n^\top$ to an element e below n can be estimated:

$$d_{e,q} \leq d_n^\top = \top_n + r_n.$$

Such bound $d_n^\top$ can be used to save distance calculations as shown in Section 4.

3.2.3 Domain-specific Text-length Heuristics

In the case of the Levenshtein edit distance we propose the use of text-length heuristics for efficiently computing both a lower and an upper bound. These heuristics are applicable in case all edit operations (insert, delete, replace) are equally weighted.

The lower-bound heuristics rely on the observation that even if for two texts of different length the shorter one is an exact substring of the longer one, the length difference must be edited by either insert or delete operations (depending on which text is longer):

$$d_{n,q} \geq \bot_{n,Length} := ||length(n) - length(q)||.$$

For the upper bound we observe that in the worst case the texts are different in each character. In that case, first all characters of the smaller length of the two texts can be replaced and the remaining characters can be either inserted or deleted (depending on which text is longer). Either way the unweighted edit distance can never be greater than the length of the longer text:

$$d_{n,q} \leq \top_{n,Length} := \max(length(n), length(q)).$$

4 Optimization of the Range-search Algorithm

A range query is the most basic similarity query explicitly specifying the query hypersphere H_Q by its center q and its query radius r_q. Due to this, the query can traverse the tree in any order (breadth first, depth first, ...) where depth first has the lowest space requirement. Independent of the expansion order for each node n it must be decided whether and how to process its child nodes. As a first attempt, in `processHeuristics` (Listing 1), the algorithm tries to process the node based only on the heuristics. If this is not possible, the algorithm falls back to computing the distance $d_{n,q}$ and continues with `processDistance` (Listing 2). In this algorithm the following optimization aspects can be applied independently.

Algorithm 1 Basic Range Query: `processHeuristics`

```
processHeuristics(node $n$,$\bot_n$,$\top_n$):
 $d^{\bot}_{n,relaxed} = \max(0, \bot_n - r_n)$

 // either determine, that no overlap is
 // possible (do nothing, return true)
 // or leave the work to
 // rangeQuery/processDistance
 // (return false)
 return $d^{\bot}_{n,relaxed} > r_q$
```

Algorithm 2 Basic Range Query: `processDistance`

```
processDistance(node $n$,$d_{n,q}$):
 if $d_{n,q} - r_n \leq r_q$:
  // overlap
  if $n$ is leaf:
   add $n$ to result set

  else:
   for each child $c$ in $n$:
    rangeQuery($c$)
```

4.1 One-child Cut

A tree of enforced equal leaf depth without balanced node splitting (like the M-Tree) sometimes tends to build "aerial roots" – i.e., chains of nodes with only one child. The efficiency of the search in such a tree is determined by the possibility to prune many subnodes at once when examining a single parent node. This advantage is gone in case a node n has just one child c. The algorithm would make some effort in examining n to determine whether it should spend even more effort to examine the child nodes. In case of only one child c it is better to examine c directly. However, care must be taken in case the centers of n and c differ, as this has effect on some heuristics on the grand children (like the parent-distance heuristics). As shown in Listing 3 we replace the examination of a node n with only one child directly by an examination of its child c without ever computing the distance to n.

4.2 Intelligent Combination of Heuristics

As discussed in Section 3.2.1 several domain-dependent and -independent heuristics can be applicable. They are grouped by their approximate run time. For example, the parent-distance heuristics and the text-length heuristics are considered to be fast, while the bag heuristics

Algorithm 3 Range Query: One-child Cut

```
rangeQuery(node $n$):
  if $n$ has exactly 1 child:
    set $c$ = the child of $n$
    rangeQuery($c$)

  else:
    ...
```

are considered slow. This grouping and the decision which heuristics to apply must be done by the user when initializing the M-Tree.

When examining a node, first the maximum of all fast lower-bound heuristics and the minimum of all fast upper-bound heuristics are computed. Using these bounds, the algorithm tries to process the node n. (For a discussion of the use of upper bounds see the subsequent sections.) If this is not possible, it tries the processing based on the slower heuristics (taking also into account the bounds computed using faster heuristics). Finally the algorithm (Listing 4) falls back to actually computing the distance.

Algorithm 4 Range Query: Combination of Heuristics

```
rangeQuery(node $n$):
  for each group in heuristicsGroups:
    $\bot_n^{group}$ = max($\bot_{n,i}$ in group, $\bot_n^{group-1}$)
    $\top_n^{group}$ = min($\top_{n,i}$ in group, $\top_n^{group-1}$)

    if processHeuristics($n$, $\bot_n^{group}$, $\top_n^{group}$):
      return

  compute $d_{n,q}$
  processDistance($n$, $d_{n,q}$)
```

4.3 Zero Interval

The easiest optimization (Listing 5) can be applied in case $\bot_n = \top_n$. If the upper and lower bound happen to be the same, we immediately know the actual distance: $d_{n,q} = \top_n (= \bot_n)$. This can be used to process the node as if the actual distance had been computed (i.e. in the best possible manner) without actually computing it.

4.4 Upper-bound Enclosure

Using the upper distance bound, we sometimes come across a situation, where

$$d_n^{\top} = \top_n + r_n \leq r_q.$$

Algorithm 5 Range Query: Zero Interval

```
rangeQuery(node $n$):
 $\bot_n$ = ...
 $\top_n$ = ...

 if $\bot_n$ == $\top_n$:
  $d_{n,q} = \bot_n$
  processDistance($n$, $d_{n,q}$)
  return

 if processHeuristics($n$, $\bot_n$, $\top_n$):
   return

 compute $d_{n,q}$
 processDistance($n$, $d_{n,q}$)
```

This means that even the furthest possible element below n is inside the query hypersphere. As shown in Listing 6, we therefore add the whole subtree below n to the result set without any further distance calculation to a predecessor (or other node examinations).

Algorithm 6 Range Query: Upper-bound Enclosure

```
processHeuristics(node $n$, $\bot_n$, $\top_n$):
 $d_n^\top = \top_n + r_n$

 if $d_n^\top \leq r_q$:
  add all data elements below $n$
      to result set
  return true

 else:
  ...
```

4.5 Upper-bound Intersection

The basic range-query algorithm tests for each examined node whether the node and the query hypersphere intersect. If so, the node is expanded, i.e., all direct child nodes are examined.

The basic algorithm uses lower bounds to test whether an intersection of node and query hypersphere is impossible. If the node cannot be pruned based on this information, the actual distance is computed. However, sometimes it is possible to preclude an intersection only using the upper bound without computing the actual distance. Node and query hypersphere definitely intersect, if

$$\top_n + r_n > r_q \geq \top_n - r_n.$$

This explicitly excludes the possibility of $r_q \geq \top_n + r_n$, which is better handled using the optimization in Section 4.4. In this case, the node n can be expanded (i.e. its child nodes are examined) without computing the distance to n. Listing 7 shows the principle.

Algorithm 7 Range Query: Upper-bound Intersection

```
processHeuristics(node $n$,$\bot_n$,$\top_n$):
 // ... first test Upper Bound Enclosure

 //now Upper Bound Intersection:
 if $\top_n + r_n > r_q \geq \top_n - r_n$:
  for each child $c$ in $n$:
   rangeQuery($c$)

  return true

 else:
  ...
```

This optimization should only be applied with care as it has some negative side effects. The parent-distance heuristics require the distance $d_{n,q}$ to the parent node n to be known when the child node c is examined. If this optimization is used, $d_{n,q}$ is not known any more. Instead we only know a lower ($\bot_n$) and upper ($\top_n$) bound for the distance to the parent node when examining the child nodes. $\bot_n$ and $\top_n$ can still be used to bound $d_{c,q}$, but the child bounds will be more loose, allowing less often to prune without computing distances. So in the worst case the optimization saves one distance computation ($d_{n,q}$) but triggers N other distance computations $d_{c_i,q}$ to all N child nodes. This negative effect can be reduced in case there are other heuristics which compensate for a less tight parent-distance heuristics. For this reason, we apply this optimization only, if other fast heuristics are available.

5 Optimization of the (k-)Nearest-Neighbor Search Algorithm

5.1 Overview

In the k-nearest-neighbor search the k closest elements to a query object q are to be found. There exists an equivalent range query whose range r_q is the distance to the furthest of the k results. (Assuming all distances – especially the kth and $k + 1$th – are not similar.) The problem is that r_q is not known in advance. However, it can be bound in the course of the search. In case k or more nodes are already examined, r_q will never be greater than the kth furthest of all examined elements. Thus, there is always a bound $\bot_{r_q}$ on r_q which shrinks during the search process. Contrary to a range query, results that lie inside a hypersphere $H_Q(q, \bot_{r_q})$ are only result candidates and not final search results. Given an oracle, which would tell the final r_q, the theoretical minimum of necessary distance computations is that of the equivalent range query, because at least the tree has to be descended using this r_q.

In case of a range query, the order of node expansion and the timing of heuristics usage do not matter. Due to the fixed query hypersphere, node examinations are independent of each other and always deliver the same result. In case of a k-nearest-neighbor query this is not the case any more. A distant node, which would not be expanded in case of an already shrunken radius, may be expanded if examined early in the search (still large radius). On the other hand, expanding many close nodes early in the search course can shrink $\perp_{r_q}$ faster than first expanding distant nodes. Hence, instead of the arbitrary order of range-query tree traversal, the k-nearest-neighbor search should examine promising nodes (with high proximity to q) first. Inverse node proximity P is the closest possible distance that a data element e below n can have to the query center q according to current knowledge. In its simplest version this boils down to $P = \max(d_{n,q} - r_n, 0)$.

5.2 Classic Algorithm

The classic algorithm of Ciaccia et al. [CPZ97] (Listing 8) uses a priority queue to expand nodes in order of their proximity. It makes, however, rather ineffective use of distance bounds. Only before inserting a node n into the priority queue the bound is used to check if n may contain elements closer than the currently known closest k results. This algorithm minimizes the number of node expansions, but not the number of distance calculations. The problem is that at the time the heuristic check is made to avoid the distance calculation $d_{c,q}$, the query radius limit $\perp_{r_q}$ is still large. It is only reduced significantly when actual data elements are added to the queue. Due to the sorting of the queue in terms of the proximity of the nodes, at this point only not-so-promising nodes are still in the queue and the search

Algorithm 8 Basic kNN Search

```
kNN(root):
  compute $d_{root,q}$ and $P_{root}$
  add root to priority queue based on $P_{root}$

  while priority queue not empty:
    remove front node $n$ from priority queue
    update $\perp_{r_q}$ based on $n$

    if $n$ is leaf:
      add $n$ to result candidates

    else:
      for each child $c$ of $n$:
        if $d_c^\perp \leq \perp_{r_q}$:
          compute $d_{c,q}$ and $P_c$
          add $c$ to priority queue based on $P_c$
```

is almost finished. Another problem is the permanent updating of $\perp_{r_q}$ which is only easy in case of $k = 1$.

5.3 Delayed Distance Computation

We present a better algorithm, which avoids computing $\perp_{r_q}$ completely and minimizes the number of distance computations to the theoretical minimum. The main idea is to delay the actual distance computation. For a node n, first an optimistic proximity $P_n^\perp$ is computed based on the lower bound $\perp_n$:

$$P_n^\perp = \max(\perp_n - r_n, 0).$$

The node is inserted into the priority queue based on $P_n^\perp$. When it is retrieved from the queue, it is the most promising node currently known. At this time the distance calculation can no longer be delayed, as we need to find out how promising the node really is. For this, the actual distance $d_{n,q}$ and the actual proximity P_n are computed. From this, there are two possible continuations:

1. n can be reinserted into the queue based on P_n to expand it (insert its children c based on $P_c^\perp$) when it is extracted again.
2. The other possibility is to expand it directly. In this case, the children c of n would be inserted directly into the priority queue, based on $P_c^\perp$.

In both cases we do not need extra distance calculations in this step, as the expansion and insertion of the children only needs some tree traversal and heuristics calculation. However, there is a downside of case 2. If expanding the children early,

- the expansion effort could be wasted, as search might be over before n must be expanded, and
- we permanently operate on a longer priority queue during search.[14]

In both cases the algorithm stops if

- the priority queue is empty or
- k elements are found and P of the currently extracted node n is greater than the worst distance of the k elements. n cannot improve the result and the remaining nodes in the queue are known to be worse. (At this point the classic algorithm would continue emptying the queue and only by the parent-distance heuristics removing child nodes.)

It can be shown [Gu16] that both variants lead to the theoretically possible minimum number of distance computations. I.e., the same distances are computed as for an equivalent range query. Listings 9 (Case 1) and 10 (Case 2) show the principle of the two cases.

[14] The effort of a single operation on such a queue is $O(\log N)$ – so all operations will be a bit more expensive.

Algorithm 9 kNN: Delayed Distance Computation (Case 1)

```
kNN(root):
 compute $d_{root,q}$ and $P_{root}$
 add root to priority queue based on $P_{root}$

 while priority queue not empty:
  remove front node $n$ from priority queue

   if $n$ was stored based on $P_n^\perp$:
    compute $d_{n,q}$ and $P_n$
    reinsert $n$ based on $P_n$

   else if $n$ is leaf:
    add $n$ to result elements using $d_{n,q}$
    if $k$ result candidates are known:
     return result candidates //we are finished

   else: // inner node
    for each child $c$ of $n$:
     estimate $\perp_c$ and $P_c^\perp$ (using $d_{n,q}$)
     insert $c$ based on $P_c^\perp$
```

5.4 Slim Radii

In [Gu16] we present another M-Tree optimization, where nodes can be shrunken to not just the enclosure of their child nodes, but to only all the data elements below. In this case it is possible that a child node n seems to have a closer proximity than its parent node p.

$$P_p > \tilde{P_n} = \max(d_{n,q} - r_q, 0).$$

Since P_n measures the closest possible distance an element below n can have to q, and all elements below n are also below p, none of them can be closer than P_p. So the maximum of all ancestor P's should be used as P_n.

5.5 Reuse of Range-query Optimization

Most of the optimizations found for range queries (see Section 4) can and should be applied to k nearest neighbors as well. For example, multiple heuristics should be combined, taking into account the computation time of the heuristics. If a node has only one child, instead of the node its child can be put on the queue directly. Also, if upper and lower bound happen to be the same, the distance can be concluded without computing it (zero interval).

Algorithm 10 kNN: Delayed Distance Computation (Case 2)

```
kNN(root):
 add root to priority queue

 set $\tilde{r}_q = \infty$
 while priority queue not empty:
  remove front node $n$ from priority queue

  if $P_n^{\perp} \geq \tilde{r}_q$:
   // no improvement possible as all
   // elements of queue are worse
   return current candidates as result

  compute $d_{n,q}$ and $P_n$
  if $P_n < \tilde{r}_q$:
   if $n$ is leaf:
    add $n$ to result candidates
    update $\tilde{r}_q$ based on $d_{n,q}$

   else:
    for each child $c$ of $n$:
     estimate $\perp_c$ and $P_c^{\perp}$ (using $d_{n,q}$)
     insert $c$ based on $P_c^{\perp}$
```

6 Experimental Results

6.1 Experimental Design

For evaluating the presented optimizations of the query algorithms, we filled a set of 100 trees (in main memory) with 10’000 elements of representative random data in different domains and queried each tree with 40’000 random queries. As domains we used:

- a range of Euclidean vector spaces (from 2-dimensional up to 15-dimensional). Those data are drawn randomly from a clustered distribution. There are 10 random cluster centers. Data within a cluster follow a gaussian normal distribution.
- the Levenshtein edit distance on sample texts. Those are drawn randomly out of a pool of 270’000 lines of programming source code.
- a similarity function on wafer-deformation patterns in the semiconductor industry. The data are taken from the lithographic step in processing wafers, where deformations have to be corrected during exposure of the wafer. As distance we use the integral of the absolute distance function over the wafer surface. The distance at one point on the wafer surface is the euclidian distance of the two deformation vectors[15].

[15] At each point the wafer has a deviation in x- and y-direction from its nominal position.

Over all trees and queries we counted the number of necessary distance calculations and computed an average per search. Since the resulting query effort varies considerably among the domains (due to the curse of dimensionality), we normalized the resulting effort in relation to the effort of a standard strategy, so that the result visualization becomes more readable.

6.2 Range Search

For range queries we compared 3 algorithms:

1. The naive algorithm without use of distance bounds ("None"),
2. The algorithm presented in [CPZ97] ("classic M-Tree") which uses distance heuristics based on the precomputed distance to the parent node (see Section 3.1.1) and
3. The combination of all new optimizations (see Section 4, "EM-Tree").

As shown in Figure 2, over all domains the parent-distance heuristics were able to save a slight amount of distance calculations. The combination of the new optimizations presented in this paper was able to further reduce the number of distance calculations per search by approximately 40% over all domains. The new algorithm is able to reduce the average range-query search effort considerably.

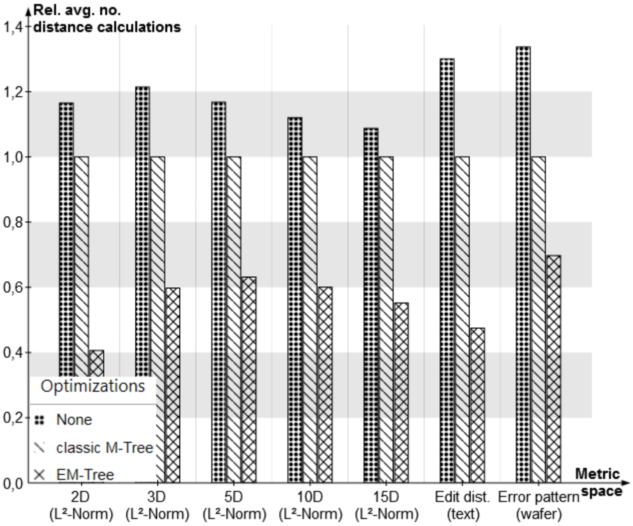

Fig. 2: Experimental Results: Range query

6.3 (k-) Nearest-Neighbor Search

For k-nearest-neighbor queries again 3 strategies were compared:

1. The naive algorithm without the use of distance bounds (“None”),
2. The algorithm presented in [CPZ97] (“classic M-Tree”) which uses distance heuristics based on the precomputed distance to the parent node (see Section 3.1.1) and
3. The combination of all new optimizations (see Section 4, “EM-Tree”).

As shown in Figure 3, over all domains the classic algorithm presented in [CPZ97] is only able to reduce the number of distance calculations a bit, because the heuristics are used in a quite inefficient manner. Using the new algorithm presented in Section 5 the number of necessary distance calculations can be further reduced significantly.

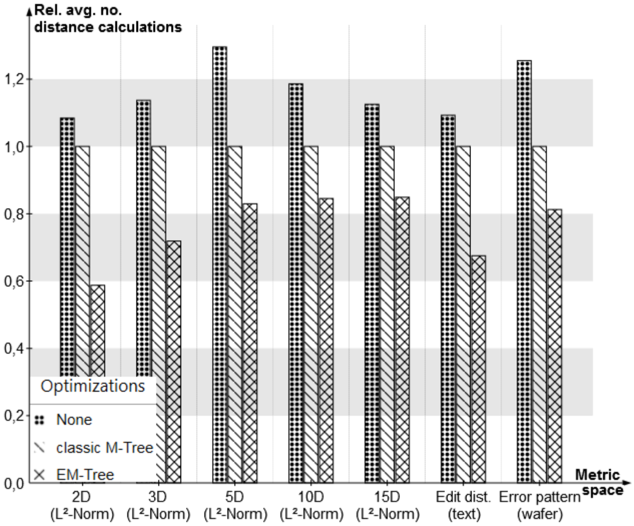

Fig. 3: Experimental Results: kNN query

7 Summary

In this paper we present new algorithms for a more efficient similarity search on the classic M-Tree [Ze06]. We identified optimization concepts to reduce the number of distance calculations which are the major part of search time (in most metric spaces even outweighing disk-access time). We then applied this approach to develop more efficient algorithms for range and k-nearest-neighbor search. The optimizations are presented in a modular fashion, so that they can be applied independently. In an experimental evaluation with a broad range of metric spaces we were able to show that these optimizations can significantly reduce the number of distance calculations in both query types.

We only implemented the tree in a prototypical manner in main memory. (This is sufficient due to the simple counting for our main goal of minimizing the number of distance calculations.) Future work would be to implement the tree in an efficient manner for use on paginated disks to really compare timings and disk accesses. Also more domains and index structures should be included in the comparison as in [La11]. Search effort depends on both the algorithms (optimized in this paper) and the tree structure. Contrary to the B-Tree, the M-Tree has degrees of freedom in its structure. We intend to explore the possibility of an optimized use of these degrees of freedom in a manner that the tree can be searched more efficiently. Further, some parameters of the search strongly depend on the domain in their availability, effort and efficiency (e.g. the order and use of different heuristics or the efficiency of single optimizations of the range query). Future work should try to explore the possibility to automatically optimize the algorithmic search (and also tree edit) parameters in accordance with the presented metric space.

References

[AS07] Aronovich, Lior; Spiegler, Israel: CM-tree: A dynamic clustered index for similarity search in metric databases. Data & Knowledge Engineering, 63(3):919–946, 2007.

[Ba94] Baeza-Yates, Ricardo; Cunto, Walter; Manber, Udi; Wu, Sun: Proximity matching using fixed-queries trees. In (Crochemore, Maxime; Gusfield, Dan, eds): Combinatorial Pattern Matching, volume 807 of Lecture Notes in Computer Science, pp. 198–212. Springer, Berlin Heidelberg, 1994.

[BCP02] Bartolini, Ilaria; Ciaccia, Paolo; Patella, Marco: String matching with metric trees using an approximate distance. In: String Processing and Information Retrieval. Springer, pp. 423–431, 2002.

[BK73] Burkhard, W. A.; Keller, R. M.: Some approaches to best-match file searching. Commun. ACM, 16(4):230–236, 1973.

[BNC03] Bustos, B.; Navarro, G.; Chávez, E.: Pivot selection techniques for proximity searching in metric spaces. Pattern Recognition Letters, 24(14):2357–2366, 2003.

[BO97] Bozkaya, T.; Ozsoyoglu, M.: Distance-based indexing for high-dimensional metric spaces. ACM SIGMOD Record, 26(2):357–368, 1997.

[Br95] Brin, S.: Near neighbor search in large metric spaces. In: Proc. Int. Conf. on Very Large Data Bases (VLDB). IEEE, pp. 574–584, 1995.

[Ci00] Ciaccia, P. and Patella, M.: The M2-tree: Processing Complex Multi-Feature Queries with Just One Index. In: DELOS Workshop: Information Seeking, Searching and Querying in Digital Libraries. 2000.

[CMN99] Chávez, E.; Marroquín, J.L.; Navarro, G.: Overcoming the curse of dimensionality. In: European Workshop on Content-based Multimedia Indexing (CBMI 99). pp. 57–64, 1999.

[CMN01] Chávez, E.; Marroquín, J.L.; Navarro, G.: Fixed queries array: A fast and economical data structure for proximity searching. Multimedia Tools and Applications, 14(2):113–135, 2001.

[CN00] Chávez, E.; Navarro, G.: An effective clustering algorithm to index high dimensional metric spaces. In: Proc. 7th Int. Symp. on String Processing and Information Retrieval. pp. 75–86, 2000.

[CP98] Ciaccia, P.; Patella, M.: Bulk loading the M-tree. In: Proc. 9th Australasian Database Conf. (ADC). pp. 15–26, 1998.

[CPZ97] Ciaccia, P.; Patella, M.; Zezula, P.: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In: Proc. Int. Conf. on Very Large Data Bases (VLDB). pp. 426–435, 1997.

[DD15] Deepak, P; Deshpande, Prasad M: Operators for Similarity Search: Semantics, Techniques and Usage Scenarios. Springer, 2015.

[DGZ03] Dohnal, V.; Gennaro, C.; Zezula, P.: Similarity join in metric spaces using eD-index. In: Proc. Int. Conf. on Database and Expert Systems Applications (DEXA). Springer, pp. 484–493, 2003.

[DN88] Dehne, F.; Noltemeier, H.: Voronoi Trees and Clustering Problems. In (Ferrate, Gabriel and Pavlidis, Theo and Sanfeliu, Alberto and Bunke, Horst, ed.): Syntactic and Structural Pattern Recognition, volume 45 of NATO ASI Series, pp. 185–194. Springer Berlin Heidelberg, 1988.

[Do03] Dohnal, V.; Gennaro, C.; Savino, P.; Zezula, P.: D-Index: Distance Searching Index for Metric Data Sets. Multimedia Tools and Applications, 21(1):9–33, 2003.

[Fe12] Fenz, Dandy; Lange, Dustin; Rheinländer, Astrid; Naumann, Felix; Leser, Ulf: Efficient Similarity Search in Very Large String Sets. In: Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM). Chania, Crete, Greece, 2012.

[Gu16] Guhlemann, Steffen: Neue Indexverfahren für die Ähnlichkeitssuche in metrischen Räumen über großen Datenmengen. PhD thesis, TU Dresden, April 2016.

[He09] Hetland, Magnus Lie: The Basic Principles of Metric Indexing. In (Coello, Carlos Artemio Coello; Dehuri, Satchidananda; Ghosh, Susmita, eds): Swarm Intelligence for Multi-objective Problems in Data Mining. Springer, Berlin, Heidelberg, pp. 199–232, 2009.

[Ja05] Jagadish, Hosagrahar V; Ooi, Beng Chin; Tan, Kian-Lee; Yu, Cui; Zhang, Rui: iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. ACM Transactions on Database Systems (TODS), 30(2):364–397, 2005.

[KM83] Kalantari, I.; McDonald, G.: A data structure and an algorithm for the nearest point problem. IEEE Transactions on Software Engineering, pp. 631–634, 1983.

[La11] Lange, Dustin; Vogel, Tobias; Draisbach, Uwe; Naumann, Felix: Projektseminar 'Similarity Search Algorithms'. Datenbank-Spektrum, 11(1):51–57, 2011.

[MOV94] Micó, M.L.; Oncina, J.; Vidal, E.: A new version of the nearest-neighbour approximating and eliminating search algorithm (AESA) with linear preprocessing time and memory requirements. Pattern Recognition Letters, 15(1):9–17, 1994.

[NBZ11] Novak, David; Batko, Michal; Zezula, Pavel: Metric index: An efficient and scalable solution for precise and approximate similarity search. Information Systems, 36(4):721–733, 2011.

[No93] Noltemeier, H. and Verbarg, K. and Zirkelbach, C.: A data structure for representing and efficient querying large scenes of geometric objects: MB* trees. In: Geometric modelling. Springer-Verlag, pp. 211–226, 1993.

[NVZ92] Noltemeier, H.; Verbarg, K.; Zirkelbach, C.: Monotonous Bisector* Trees – a tool for efficient partitioning of complex scenes of geometric objects. Data Structures and Efficient Algorithms, pp. 186–203, 1992.

[NZ14] Novak, David; Zezula, Pavel: Rank aggregation of candidate sets for efficient similarity search. In: Proc. Int. Conf. on Database and Expert Systems Applications (DEXA). Springer, pp. 42–58, 2014.

[Pa99] Patella, Marco: Similarity search in multimedia databases. Dipartmento di Elettronica Informatica e Sistemistica, Bologna, 1999.

[Rh10] Rheinländer, Astrid; Knobloch, Martin; Hochmuth, Nicky; Leser, Ulf: Prefix tree indexing for similarity search and similarity joins on genomic data. In: Proc. Int. Conf. on Scientific and Statistical Database Management (SSDBM). Springer, pp. 519–536, 2010.

[Sh77] Shapiro, Marvin: The choice of reference points in best-match file searching. Communications of the ACM, 20(5):339–343, 1977.

[Sk03] Skopal, T.; Pokornỳ, J.; Krátkỳ, M.; Snášel, V.: Revisiting M-tree building principles. In: Advances in Databases and Information Systems. Springer, pp. 148–162, 2003.

[Sk04] Skopal, T.: Pivoting M-tree: A metric access method for efficient similarity search. In: Proc. Dateso Annual Int. Workshop on Databases, Texts, Specifications and Objects (Desna, Czech Republic, April 14-16). pp. 27–37, 2004.

[SMZ15] Sedmidubsky, Jan; Mic, Vladimir; Zezula, Pavel: Face Image Retrieval Revisited. In: Proc. Int. Conf. on Similarity Search and Applications. Springer, pp. 204–216, 2015.

[Tr00] Traina, C.; Traina, A.; Seeger, B.; Faloutsos, C.: Slim-trees: High performance metric trees minimizing overlap between nodes. Advances in Database Technology, EDBT 2000, pp. 51–65, 2000.

[Tr02] Traina Jr, C.; Traina, A.; Faloutsos, C.; Seeger, B.: Fast indexing and visualization of metric data sets using slim-trees. Knowledge and Data Engineering, IEEE Transactions on, 14(2):244–260, 2002.

[Uh91] Uhlmann, J.K.: Satisfying general proximity/similarity queries with metric trees. Information processing letters, 40(4):175–179, 1991.

[Vi86] Vidal, E.: An algorithm for finding nearest neighbours in (approximately) constant average time. Pattern Recognition Letters, 4(3):145–157, 1986.

[Yi93] Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: Proc. 4th Annual ACM-SIAM Symp. on Discrete Algorithms. Society for Industrial and Applied Mathematics, pp. 311–321, 1993.

[Yi99] Yianilos, P.N.: Excluded middle vantage point forests for nearest neighbor search. In: Proc. 6th DIMACS Implementation Challenge: Near Neighbor Searches (ALENEX). Baltimore, Maryland, USA, 1999.

[Ze06] Zezula, Pavel; Amato, Giuseppe; Dohnal, Vlastislav; Batko, Michal: Similarity Search: The Metric Space Approach. Springer, Berlin, 2006.

[Zh05] Zhou, X.; Wang, G.; Zhou, X.; Yu, G.: BM+-tree: A hyperplane-based index method for high-dimensional metric spaces. In: Database Systems for Advanced Applications. Springer, pp. 398–409, 2005.

[ZS15] Zierenberg, Marcel; Schmitt, Ingo: Optimizing the Distance Computation Order of Multi-Feature Similarity Search Indexing. In: Proc. Int. Conf. on Similarity Search and Applications. Springer, pp. 90–96, 2015.