Organizing Similarity Spaces using Metric Hulls*

Miriama Jánošová[1], David Procházka[1], and Vlastislav Dohnal[1]

Faculty of Informatics, Masaryk University, Brno, Czech Republic
{x424615,xprocha6,dohnal}@fi.muni.cz

Abstract. A novel concept of a metric hull has recently been introduced to encompass a set of objects by a few selected border objects. Following one of the metric-hull computation methods that generate a hierarchy of metric hulls, we introduce a metric index structure for unstructured and complex data, a Metric Hull Tree (MH-tree). We propose a construction of MH-tree by a bulk-loading procedure and outline an insert operation. With respect to the design of the tree, we provide an implementation of an approximate kNN search operation. Finally, we utilized the Profimedia dataset to evaluate various building and ranking strategies of MH-tree and compared the results with M-tree.

Keywords: metric-hull tree · metric hull · index structure · nearest-neighbor query · similarity search

1 Introduction

Content-based retrieval systems have become often applied to complement traditional retrieval systems. Such systems allow processing complex data, such as photos, medical images, protein sequences or audio recordings, and support similarity queries. Such search requests compare data items based on the similarity of their content or descriptors extracted from the content rather than the identity of data. The challenge is managing the ever-growing complex data efficiently and evaluating the similarity queries faster than by the sequential scan. Many indexing structures were proposed ranging from clustering-based ones [7, 22], space-partitioning methods [6, 4] to transformation techniques [1].

Complex data are thus expressed as descriptors capturing important features from their content, e.g., color histogram, texture, shape [21] or more profound vectors computed by convolutional networks [10]. Thus, the descriptors are often high-dimensional spaces[1] [18]. The problem of *dimensionality curse* then arises [5]. It leads to visiting many data partitions by an index due to frequent overlaps among them, whereas useful information is contained in a few of them. So the index must employ further filtering constraints to make query evaluation efficient [20, 12].

* The publication of this paper and the follow-up research was supported by the ERDF "CyberSecurity, CyberCrime and Critical Information Infrastructures Center of Excellence" (No.CZ.02.1.01/0.0/0.0/16_019/0000822).

[1] Or even distance spaces where no implicit coordinate system is defined.

A novel concept of metric hulls has been introduced recently [2]. The purpose of the metric hull is to embrace a set of metric objects. The metric hull is defined as a set of objects selected out of the set to encompass. We build upon this concept to create a hierarchical search structure where a metric hull represents each node. Since the authors also provide a test of whether a metric object is part of the hull or not, such a structure is viable. We perceive the metric hull as an alternative to the metric ball used by M-tree [7] or Slim-tree [22]. However, it bounds data much tighter without any additional information. As a result, node overlaps can be reduced. The issue of intersecting balls surrounding Voronoi cells is studied in VD-tree [13].

This paper proposes a metric access method that organizes data in metric hulls and addresses the issue of large node overlaps without the need for external pivots, as was applied in Pivoting M-tree [19]. We take advantage of algorithms to construct metric hulls incrementally [2] to build a hierarchy of metric hulls. Next, the issue of comparing and ordering metric hulls with respect to a similarity query is studied here in this paper. We test different variants of such and evaluate the performance of approximate k-nearest neighbors search.

The remaining parts of the paper are structured as follows. In the next section, there is a concise summary of metric space indexing and similarity queries, and more importantly the concept of metric hulls. Related work of indexing structures is surveyed in Section 3. The core of this paper is the proposal of Metric Hull Tree, presented in Section 4. Performance evaluation on a real-life high-dimensional data is described in Section 5. Contributions of this paper and possible future extensions are summarized in the last section.

2 Preliminaries

A *metric space* M is a pair $\mathcal{M} = (\mathcal{D}, d)$, where $\mathcal{D}$ is a domain of objects, and d is a distance function (metric) $d : \mathcal{D} \times \mathcal{D} \to \mathbb{R}_0^+$ satisfying metric postulates, namely non-negativity, the identity of indiscernibles, symmetry, and triangle inequality. A set of data objects to be queried, so-called *database*, is denoted as $X \subseteq \mathcal{D}$.

We distinguish two common retrieval operations, specifically, the *range query* ($range(q, r)$) – returning database objects, such that their distance to q is smaller than the distance r; and the *k-nearest neighbors query* ($k\mathrm{NN}(q)$) – retrieving k database objects closest to the query object q; when there are more objects at the distance of the k-th nearest neighbor, the ties are solved arbitrarily. Nowadays, approximate evaluation of similarity queries (e.g., approximate $k\mathrm{NN}(q)$) loosens the restrictions on returning the genuine answer at much lower search costs. Such evaluation can be implemented by an early termination strategy that stops the search when a predefined number of data objects are visited. The identification of the most relevant data parts is thus the center of interest.

A more advanced data processing technique is the Similarity Group By (SGB) query [11]. It groups data by respecting similarity constraints, e.g., distance threshold. However, the disadvantage of such queries is that the obtained groups are mere lists of objects. Thus, there is no compact representation of such groups.

Hence, the objective of [2] was to examine properties of objects' groups' representations, where the *hull representation* proved to be the most compact.

2.1 Hull Representation

Let C be a group of objects from database $C \subseteq X$. Formally, the **hull representation** [2] is defined as $\mathcal{H}(C) = \{p_i \mid p_i \in C\}$ and any other object $o \in C$ is **covered** by hull. Each p_i corresponds to a boundary object of C referred to as *hull object*.

Let $\mathcal{H}$ be a hull representation $\mathcal{H} = \{p_1, \ldots, p_h\}$ and an object $o \in \mathcal{D}$. Assume p_{NN} to be the nearest hull object of $\mathcal{H}$ to o, i.e., $NN = \text{argmin}_{i=1..h}(d(o, p_i))$. We say the object o is **covered** by $\mathcal{H}$ if and only if

$$\sum_{i=1..h, i \neq NN} d(p_i, o) \leq \sum_{i=1..h} d(p_i, p_{NN}). \tag{1}$$

By the original definition, the smallest hull consists of three objects. If $|\mathcal{H}| < 3$, the hull objects are the only objects covered.

Antol et al. [2] proposed two algorithms for hull computation for a set of objects C. First, the *Basic Hull Algorithm* starts with selecting the furthest object in O and gradually adds additional objects that are furthest objects from the already selected ones. E.g., the third object has the maximum sum of distances to the previous two. This procedure terminates when the whole set C is covered by $\mathcal{H}$. In the worst case, each $o \in C$ becomes a hull object. Second, the *Optimized Hull Algorithm* is an improvement to the basic one, which reduces the number of hull objects. After selecting the initial three hull objects, the procedure is modified: instead of adding the furthest not-yet-covered object o_f to $\mathcal{H}$, the algorithm tries to replace some existing hull object with o_f to increase the coverage of C. This leads to fewer hull objects without compromising the fact that each object of C is covered by the resulting $\mathcal{H}$.

3 Existing Metric Access Methods

In this section, we overview existing metric indexes relevant to this work. We start with structures organizing objects into metric balls. The first disk-oriented and dynamic structure built in the bottom-up fashion is the M-tree [7]. Data objects are grouped into leaf nodes that are, in turn, represented by metric balls (i.e., a routing object and a covering radius). Further levels group metric balls into larger ones ending with l entries in the root node. The disadvantage is major overlaps among such ball regions. A slim-down algorithm in Slim-trees [22] has later optimized such an issue. The tree compactness is measured by the fat factor there. Additional objects are included in internal nodes to further split balls into hyperplanes in M$^+$-tree [25] and BM$^+$-tree [26]. Pivoting M-tree [19] selects a fixed set of pivots that are globally used to define ranges on distances within which objects reside – spherical cuts. This resembles Linear AESA principle [24],

which recomputes distances to fixed pivots and stores them in arrays to fast array-range filtering.

The other methods partition the data space by hyperplanes. GH-tree [23, 3] is the binary hyper-plane tree that was later generalized to recursive Voronoi tree, call GNAT [6]. The dynamic version is EGNAT [15], which bulk-loads the tree and then allows minor updates. Since metric balls provide a simple yet efficient way of filtering tree branches, they were incorporated into Voronoi diagrams in NOBH-tree [16]. A common disadvantage of Voronoi-diagram-based indexes is the difficulty of redefining the partitioning at reasonable costs when the tree becomes unbalanced. This is tackled in VD-tree [13]. The objects are swapped between Voronoi cells to reduce overlaps, which is analogous to the slim-down algorithm [22].

The concept of metric ball regions is widely used and proved advantageous when combined with Voronoi partitioning or pivot-based filtering (Linear AESA). This paper exploits the brand new proposal of metric hulls that can bound a set of objects tightly by outliers. Metric hulls are an alternative to selecting external pivots and consequential definition of constraints on distances for each individual tree nodes.

4 MH-Tree – the Proposed Method

This section describes the proposed Metric Hull Tree (MH-tree) that represents data partitions by metric hulls. The hulls are constructed bottom-up by following the grounds of *Incremental Hull Algorithm* [9]. Literally, it gradually merges hulls until only one final hull representation is obtained. However, the original merging procedure needs to be generalized to support larger arity than two and any capacity of leaf buckets.

4.1 Structure and Bulk Loading

The MH-tree is a hierarchical tree structure composed of two node types, as depicted in Figure 1. Each *Leaf node* encapsulates a bucket – a storage of $[c, 2c]$ objects. Each leaf node is rooted for an internal node and represented there by a metric hull. The *Internal node* manages up to a pairs of a hull representation ($\mathcal{H}_i$) and a pointer (ptr_i) to a leaf node. Each hull is constructed by calling the *Optimized Hull Algorithm*, see Section 2.1 and [2].

We construct the MH-tree by a bulk-loading procedure. Firstly, we group the database objects into leaf nodes containing c objects. Secondly, a closest leaf nodes are merged, thus obtaining a level of internal nodes. This merging is repeated until one node is obtained, becoming the root of the MH-tree. This procedure creates a balanced a-ary tree. We present it in pseudo-code in Alg. 1. If there are too few objects (incapable of forming at least two leaves), we create just one leaf node that forms the root. In the following, we detail the sub-algorithms.

The database X is clustered by forming compact clusters of c objects. Thus, $\lfloor \frac{|X|}{c} \rfloor$ leaf nodes are created. Algorithm 2 presents the pseudo-code of CreateLeafNodes. The procedure starts with selecting the furthest object o_f from a

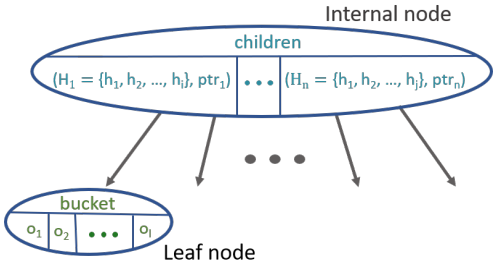


Fig. 1: A schema of MH-tree structure.

Algorithm 1: BULKLOAD(X, c, a)

Input: a database X, bucket capacity c, arity a
Output: a root node of MH-tree

```
1  if |X| < 2c then
2      root ← create a new leaf node;
3      root.H ← compute Optimized Hull representation of X;
4      root.bucket ← X;
5      return root;
6  nodes ← CREATELEAFNODES(X, c);
7  while |nodes| ≠ 1 do
8      nodes ← MERGENODES(nodes, a);
9  root ← nodes[0];
10 return root;
```

random object in X (Lines 7-8), a new cluster's nucleus. The second object in the cluster is the nearest neighbor of o_f. To add the next object, we execute 1NN queries for each object already assigned to the cluster and choose the neighbor that minimizes the sum of distances to objects already in the cluster. This is repeated until the cluster contains c objects (Lines 9-12). The next cluster is formed by analogy but omitting the already clustered objects. If there are fewer than c unprocessed objects, they get assigned to closest clusters directly, i.e., some clusters can contain more than c objects. Finally, the leaf nodes storing the clusters' objects in buckets are returned.

The motivation of Alg. 2 is to create compact clusters of up to $2c$ objects also for data with outliers and/or overlapping clusters. Here, agglomerative clustering linking closest objects/clusters would create much more overlaps among hulls. In particular, whenever a cluster exceeds c objects, it is taken out and forms a leaf node. Consequently, the remaining objects would very likely be outliers. They would group to a hull that would span over all the other nodes.

The next bulk-loading stage is merging leaf nodes to create a balanced structure of internal nodes. It is specified in Alg. 3. By analogy we start with the furthest leaf node (n_f) and execute the aNN(n_f) query to get a cluster of a near

Algorithm 2: CreateLeafNodes(X, c)

Input: a database X, bucket capacity c
Output: a set of leaf nodes

```
1  leafNodes ← ∅;
2  while X ≠ ∅ do
3      if |X| < c then
4          foreach o ∈ X do
5              add o to the closest node in leafNodes;
6          break
7      o_f ← the furthest object in X for a randomly picked object from X;
8      X ← X \ o_f; cluster ← { o_f };
9      while |cluster| < c do
10         NNs ← { o_n | ∃o_c ∈ cluster : o_n ∈ 1NN(o_c) };
11         o ← argmin_{o_n ∈ NNs} Σ_{o_c ∈ cluster} d(o_n, o_c);
12         X ← X \ o; cluster ← cluster ∪ { o };
13     leaf ← create a new leaf node;
14     leaf.bucket ← cluster;
15     leafNodes ← leafNodes ∪ { leaf };
16 return leafNodes;
```

leaf nodes, n_f inclusively. These nodes form an internal node for the next tree level. We repeat this procedure until all nodes are processed. The identification of furthest and close nodes is based on a comparison of nodes' hulls. We consider the distance between hulls $\mathcal{H}_1$ and $\mathcal{H}_2$ to be defined as:

$$d(\mathcal{H}_1, \mathcal{H}_2) = \min_{\forall h_1 \in \mathcal{H}_1, \forall h_2 \in \mathcal{H}_2} d(h_1, h_2). \tag{2}$$

So, the furthest node is thus the node whose hull is furthest from the hull of a randomly picked node (out of not-yet-processed ones). The outcome of Alg. 3 is a list of nodes constituting the next level of MH-tree. We apply it until only one node is returned – the root node.

To create the hull representations we utilize the Optimized Hull Algorithm (called from Algorithm 3). When merging leaf nodes, the Optimized Hull Algorithm is invoked on the objects of the leaf node's bucket to obtain a hull. In this course, we would collect all objects from the previously merged nodes to create a hull. But the computational requirements would grow steeply then. Instead, in the next generations we gather the hull objects from all hulls in the internal node and compute the new hull on them solely. This practice introduces imprecision of hulls – some objects stored in the sub-tree may not be covered. We address this issue on the kNN search algorithm.

Algorithm 3: MERGENODES(N, a)

Input: a set of nodes N (at the same level of the tree), arity a
Output: a set of internal nodes of the upper level

$level \leftarrow \emptyset$; $notProc \leftarrow N$;
while $notProc \neq \emptyset$ **do**
  if $|notProc| \leq a$ **then**
    create a new internal node $nNode$;
    foreach $node \in notProc$ **do**
      $\mathcal{H} \leftarrow$ call Optimized Hull Algo on $node$;
      $ptr \leftarrow$ pointer to $node$;
      $nNode.HullChildPairs \leftarrow nNode.HullChildPairs \cup \{ (\mathcal{H}, ptr) \}$;
    $level \leftarrow level \cup \{ nNode \}$;
    break
  $n_f \leftarrow$ extract the furthest node from $notProc$;
  $C \leftarrow$ execute $(a-1)$NN(n_f) query on $notProc$;
  $notProc \leftarrow notProc \setminus C$;
  create a new internal node $nextNode$;
  foreach $node \in C \cup \{ n_f \}$ **do**
    $\mathcal{H} \leftarrow$ call Optimized Hull Algo on $node$;
    $ptr \leftarrow$ pointer to node;
    $nextNode.HullChildPairs \leftarrow nextNode.HullChildPairs \cup \{ (\mathcal{H}, ptr) \}$;
  $level \leftarrow level \cup \{ nextNode \}$;
return $level$;

4.2 Searching in the MH Tree

We outline the kNN search algorithm in Alg. 4. We assume a limit on the number of visited data objects is passed, so the STOP function can terminate the search early. Such limitation together with the imprecisions introduced during the building procedure result in acquiring an approximate result. The algorithm starts from the root node and maintains a queue of nodes to be inspected. This queue is ordered by the "likelihood" of the node to contain relevant data. It can be defined as a lower bound or upper bound distance for the query object q to the nearest/furthest object in a node's hull $\mathcal{H}$. We define it using the hull objects exclusively, so the lower and upper bound are defined as

$$d_l(q, \mathcal{H}) = \min_{\forall h \in \mathcal{H}} d(q, h); \tag{3}$$

$$d_u(q, \mathcal{H}) = \max_{\forall h \in \mathcal{H}} d(q, h). \tag{4}$$

The actual definition of RANK function is investigated in the experiments in Section 5.

The exact evaluation of the kNN query can be obtained by setting the approximate limit to 100%. Even though this being a straightforward solution, it leads to scanning the whole database. Rather, the check comparing the distance

Algorithm 4: ApproximateKNNSearch(q, k, Rank, *limit*)

Input: a query object q, number of nearest neighbors k, rank function Rank, approximation parameter *limit*
Output: a set of nearest neighbors found

```
1  answer ← ∅; // ordered set by objects' distances from q
2  PQ ← create a priority queue with the priority determined by RANK;
3  insert the root node into PQ with zero priority;
4  while PQ is not empty do
5      // early termination after a certain percentage of visited objects
6      if STOP(limit) then
7          break
8      node ← extract the node with highest priority from PQ;
9      if node is a leaf node then
10         foreach o ∈ node.bucket do
11             if |answer| < k then
12                 add o into answer;
13                 continue;
14             o_k ← the k^th object from query object q in answer;
15             if d(q, o) < d(q, o_k) then
16                 insert o in answer and remove o_k from answer;
17     else
18         foreach pair ∈ node.HullChildPairs do
19             insert pair.ptr into PQ with the priority RANK(q, pair.H, k);
20 return answer;
```

to the k^{th} nearest-neighbor candidate and the distance to the hull of PQ's head element must be defined. Since the current bulk-loading algorithm does ensure coverage such a test may not be ensuring result correctness. Our primary aim in this paper is to show the viability of the application of metric hulls to indexing, and we do not study the exact evaluation. A promising direction to define a more efficient exact-search algorithm is presented in [8]. The author exploits transformation of a metric space by multiple pivots, and defines a constraint on distance of an object in such a pivot space.

4.3 Dynamicity

After bulk-loading the MH-tree, we can insert new objects to corresponding leaf nodes. Firstly, we locate the most suitable leaf $\mathcal{N}_{bl}$ by executing kNN search algorithm with the newly inserted object as the query object (Alg. 4). When a leaf node is extracted from the search queue (Line 8), we stop the search and insert the new object o into the leaf node's bucket. Next, if o is not covered by the node's hull $\mathcal{H} = \{h_1, \ldots, h_l\}$, it is updated with linear costs: $\forall i$, try to replace h_i with o and test whether h_i is covered by such updated hull. If so, this

hull is stored in the leaf's parent. Otherwise, o is added as a new hull object, i.e., the new hull $\mathcal{H} \cup \{o\}$ is stored. The hulls of parent nodes need not be updated, since the closeness of hulls to an object is addressed in the tree traversal (by the RANK function).

If the capacity of the leaf $\mathcal{N}_{bl}$ is double exceeded, we split the leaf into two new nodes $\mathcal{N}_1$ and $\mathcal{N}_2$ by these steps: (i) we identify an outlier in $\mathcal{N}_{bl}$'s bucket (the furthest object from a random one), and halve the bucket's objects (incl. o) according to the distance from the outlier – $c + 1$ objects closest to the outlier from the bucket of $\mathcal{N}_1$ and the remaining objects are stored in the bucket of $\mathcal{N}_2$. The new nodes are linked to $\mathcal{N}_{bl}$'s parent if there is room for them. Otherwise, a new internal node is created and roots the new leaf nodes. So, the MH-tree becomes unbalanced then. If the structure becomes highly unbalanced, it should be rebuilt from scratch.

5 Experimental Evaluation

This section provides an experimental study of the proposed Metric Hull Tree and compares its performance with M-tree with Slim-down.

We used the Profimedia [14] dataset in the experiments. It consists of a series of 4096-dimensional vectors extracted from Photo-stock images using a convolutional neural network. To measure the similarity between the data, we utilize the Euclidean distance. Our experiments were executed upon two sizes of Profimedia dataset – 10,000 and 100,000 objects.

To compare the performance, we execute kNN queries and measure the acquired recall. Specifically, we employ the approximate kNN search with an early termination strategy of visiting a certain percentage of database objects. The evaluation starts with the approximation parameter set to 5%, continued by a gradual increase of it in five percentage-point steps up to 100%. To quantify the trade-off between the accuracy and the efficiency of the approximate search, we compute $recall = \frac{|S \cap S_a|}{S}$, where the set S corresponds to the result of (precise) kNN query and set S_a to the result of approximate kNN query.

The pivotal part of effective search in MH-tree is the selection of the ranking function in priority queues. Naturally, we exploit just hull objects to define the ranking function. In the following sections, we study the influence of various ranking functions on recall in a shallow tree configuration, where we examine the ranking of the leaf nodes solely, and in deep tree configuration, where we take into consideration the ranking of both internal and leaf nodes. Lastly, we compare the best definition of ranking function in MH-tree with M-tree.

5.1 Ordering Leaf Nodes

We analyse the way of ordering leaf nodes by comparing their hulls to an object, e.g., a query object. The most encouraging way uses the distance from the query object to the nearest hull object. Formally, the rank of a leaf node is defined as follows

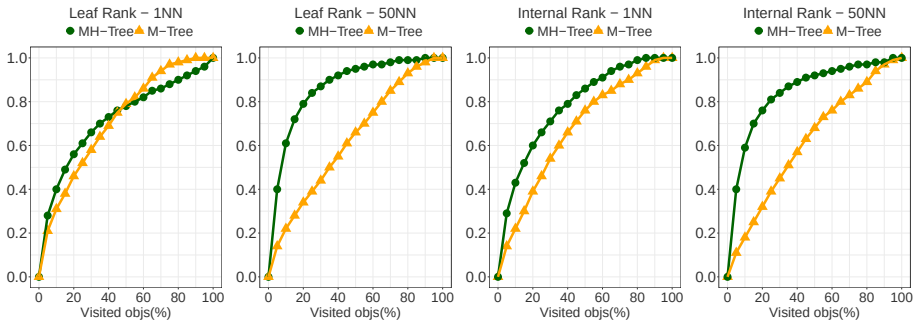


Fig. 2: Average recall of MH-tree and M-tree for 1NN and 50NN queries: (a,b) ranking leaf nodes in shallow structures, and (c,d) final ranking of both types of nodes in hierarchical structures. All results are on 10,000 Profimedia images.

$$\text{RANK}_{LEAF}(q, \mathcal{H}) = \min_{\forall h \in \mathcal{H}} d(q, h). \tag{5}$$

To determine the efficiency of the $rank_{LEAF}$ function, we evaluate the shallow configurations of MH-tree as well as M-tree, i.e., the root node referencing all leaf nodes. In Figure 2 ((a) and (b)), we provide the comparison for 1NN and 50NN queries varying the approximation limit. Regarding the 1NN query (Figure 2a), the MH-tree surpasses the performance of the M-tree in the first 50% of visited objects. After that, the M-tree manages to gather faster growth of recall. However, the performance of both approaches is more or less the same.

On the contrary, the average recall of the 50NN query (Figure 2b) reveals a much better performance of the MH-tree. It reaches 80% recall while visiting 20% of the database only. The M-tree manifests almost linear growth.

We also tested another variant of leaf-node ranking functions, e.g., the distance to the furthest hull object and average distance to all hull objects, but this nearest variant performed the best. All results are available in the bachelor's thesis [17].

5.2 Ordering Internal Nodes

To efficiently traverse also deep MH-tree structures, i.e., the multi-level ones, we need to determine the best-ranking strategy with respect to both leaf and internal nodes. Firstly, we examined ordering based solely on the distance between the query and the node without taking into consideration whether or not the query is covered by the node's hull representation. Such ranking roughly corresponds to RANK_{LEAF}. However, we experienced only a linear growth of recall to the number of visited nodes for the 50NN query. Thus, to improve navigation, the rank used in the priority queue needs to be more sophisticated.

Table 1: The definition of the most efficient rank function $\text{RANK}(q, \mathcal{H}, k)$.

Conditions			**Value**
Node type	k	*covered*	$\text{RANK}(q, \mathcal{H}, k)$
Internal	1	YES	$-\max_{\forall h \in \mathcal{H}} d(q, h)$
		NO	$\max_{\forall h \in \mathcal{H}} d(q, h)$
	> 1	YES	$-\min_{\forall h \in \mathcal{H}} d(q, h)$
		NO	$\min_{\forall h \in \mathcal{H}} d(q, h)$
Leaf			$\min_{\forall h \in \mathcal{H}} d(q, h)$

The $\text{RANK}(q, \mathcal{H}, k)$ function, defined in Table 1, computes the rank depending on whether or not the query object is covered by the hull and also on the number of neighbors k to be retrieved. We included the ranking of leaves there for completeness. For 1NN queries, it is more convenient to prefer hulls that are closer overall, so the furthest hull object is used. Whereas the nearest one is the best performing for any $k > 1$, since we do not know how many objects are present there in advance. Notably, the covered condition provides a better ordering of hulls that contain the query object, so hulls with q more to its center are preferred. More details are in the bachelor's thesis [17].

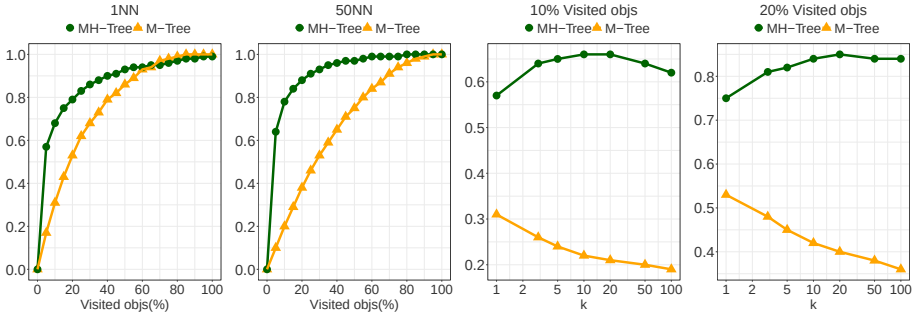
Figure 2c presents the average recall of the MH-tree and M-tree in the 1NN query. Notice that the recall has almost doubled per the same amount of visited nodes compared to Figure 2a. Therefore, the *rank* is able to reflect the performance of RANK_{LEAF} while proving that it is also able to navigate the tree effectively.We observe similar behavior when comparing Figs. 2d and 2b on 50NN queries. The MH-tree achieves significantly better recall than M-tree. The difference in MH-tree's performance on 50NN when being shallow or deep is marginal, proving the node navigation by the rank function is robust.

5.3 Comparison

We compare the MH-tree with M-tree on 100,000 objects from Profimedia. The M-tree was built with the slim-down algorithm [22] to make it as efficient and compact as possible. To validate the quality and performance of the structures, we set the structure parameters to obtain similar trees. They are summarized in Table 2. We report the values of fat-factor [22] that quantifies overlaps of covering regions representing tree nodes. The fat factor is a relative quantity computed as an average performance of zero-radius range queries for each database object. If the search for an object visits exactly one node per level, the fat factor is zero. In the worst case, all tree nodes are visited. The fat factor then grades the tree with one.

Table 2: Features of MH-tree and M-tree build on 100,000 Profimedia images.

	Params		Building statistics					
	Arity	Leaf cap.	Height	Internal nodes	Leaf nodes	routing objects	Fat factor	Building time (s)
MH-tree	100	100	2	11	1000	3262	0.03	1548
M-tree	100	200	3	51	2546	2596	0.56	67



(a) (b) (c) (d)

Fig. 3: Average recall of MH-tree and M-tree in 1NN and 50NN queries in Deep tree structure (a),(b). Average recall of MH-tree, M-tree for varying k in the kNN queries (c),(d). All results are on 100,000 Profimedia images.

In Figs. 3a and 3b, we summarize the average recall of MH-tree compared to M-tree in 1NN and 50NN queries. The recall of MH-tree rises much steeper than M-tree's up to visiting 15% of the database. For example, MH-tree provides 88% recall for 50NN queries compared to 29% of M-tree. Figures 3c and 3d showcase the details on performance of various kNN queries when approximation is fixed to 10% and 20% of dataset. The recall of M-tree deteriorates with increasing k. MH-tree's recall is more steady. The results manifest that the MH-tree is able to outperform the M-tree even on large datasets significantly.

The negative point is the construction costs that are quite high and can only be amortized when managing mostly static data. We did not focus on optimizing the building routine, but the concept of hulls can eliminate node overlaps to a large extent.

6 Conclusions

MH-tree is an index structure build upon the novel concept of metric hulls. We proposed algorithms for building a hierarchy of metric hulls that organizes data objects into leaf nodes, which are gradually merged into internal nodes constrained by metric hulls. In addition to such a bulk-loading procedure, we outlined the dynamic insertion of new objects. The fat factor of MH-tree is by one order of magnitude smaller than of M-tree with slimming-down. This proves

the compactness of metric hull representation. Admittedly, this can also be accounted to the building process of MH-tree that groups close objects primarily.

We proposed and analysed a node-ranking function that orders nodes by their closeness to a query object. The bases of leaf and internal nodes' ranking differ – the distance to the closest hull object is taken as the measure for leaf nodes. In contrast, the distance to the furthest hull object is the means for internal nodes. We also showed that coverage of the query object by a hull needs to be employed in order to navigate deeper tree structures effectively. In addition, we achieved the highest recall when distinguishing between retrieval of one neighbor and multiple nearest neighbors.

Finally, we compared the best-performing setup of MH-tree with the M-tree built by the slim-down algorithm. The results showcase that MH-tree outperforms M-tree significantly – fewer nodes are visited for the same recall or vise versa. Specifically, the performance of MH-tree was higher by 20-30% on average compared to M-tree per the same amount of visited objects on smaller datasets. The differences were even more pronounced on larger data, 30-40% higher average recall of MH-tree depending on the number of extracted neighbors.

The future work would focus on generating representations with more hull objects, thus keeping the hulls even more compact. This could result in a significant improvement in approximate search. In addition, formulating mature ranking strategies could lead to a finer-grained tree traversal. Lastly, we will compare MH-tree with techniques using external pivots, e.g., Pivoting M-tree.

References

1. Amato, G., Gennaro, C., Savino, P.: MI-File: Using inverted files for scalable approximate similarity search. Multimedia Tools and Applications (2014). https://doi.org/10.1007/s11042-012-1271-1
2. Antol, M., Janosova, M., Dohnal, V.: Metric hull as similarity-aware operator for representing unstructured data. Pattern Recognition Letters, Available online on June 12, 2021 pp. 1–8 (2021). https://doi.org/10.1016/j.patrec.2021.05.011
3. Batko, M.: Distributed and scalable similarity searching in metric spaces. In: Current Trends in Database Technology - EDBT 2004 Workshops. pp. 44–53. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
4. Batko, M., Dohnal, V., Zezula, P.: M-grid: Similarity searching in grid. In: P2PIR 2006: International Workshop on Information Retrieval in Peer-to-Peer Networks (2006). https://doi.org/10.1145/1183579.1183583
5. Böhm, C., Berchtold, S., Keim, D.A.: Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. **33**(3), 322–373 (Sep 2001). https://doi.org/10.1145/502807.502809
6. Brin, S.: Near Neighbor Search in Large Metric Spaces. Proceedings of the International Conference on Very Large Data Bases (1995)
7. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB). pp. 426–435. Morgan Kaufmann (1997)
8. Hetland, M.L.: Comparison-based indexing from first principles. arXiv preprint arXiv:1908.06318 (2019)

9. Jánošová, M.: Representing Sets of Unstructured Data. Master thesis, Masaryk University, Faculty of Informatics (2020), https://is.muni.cz/th/vqton/
10. Krizhevsky, A., Sutskever, I., Hinton, G.E.: ImageNet classification with deep convolutional neural networks. Communications of the ACM (2017). https://doi.org/10.1145/3065386
11. Laverde, N.A., Cazzolato, M.T., Traina, A.J., Traina, C.: Semantic similarity group by operators for metric data. In: International Conference on Similarity Search and Applications. pp. 247–261. Springer (2017)
12. Mic, V., Novak, D., Zezula, P.: Binary sketches for secondary filtering. ACM Trans. Inf. Syst. **37**(1), 1:1–1:28 (2019). https://doi.org/10.1145/3231936
13. Moriyama, A., Rodrigues, L.S., Scabora, L.C., Cazzolato, M.T., Traina, A.J.M., Traina, C.: VD-Tree: How to build an efficient and fit metric access method using voronoi diagrams. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing (SAC). p. 327–335. ACM, New York, NY, USA (2021)
14. Novak, D., Batko, M., Zezula, P.: Large-scale image retrieval using neural net descriptors. In: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval. pp. 1039–1040. ACM (2015)
15. Paredes, R.U., Navarro, G.: EGNAT: A fully dynamic metric access method for secondary memory. In: 2nd International Workshop on Similarity Search and Applications, SISAP 2009 (2009). https://doi.org/10.1109/SISAP.2009.20
16. Pola, I.R.V., Traina, C., Traina, A.J.M.: The NOBH-tree: Improving in-memory metric access methods by using metric hyperplanes with non-overlapping nodes. Data and Knowledge Engineering (2014). https://doi.org/10.1016/j.datak.2014.09.001
17. Procházka, D.: Indexing structure based on metric hulls. Bachelor thesis, Masaryk University, Faculty of Informatics (2021), https://is.muni.cz/th/jk21s/
18. Samet, H.: Foundations of Multidimensional And Metric Data Structures. The Morgan Kaufmann Series in Data Management Systems, Morgan Kaufmann (2006)
19. Skopal, T., Pokorný, J., Snasel, V.: PM-tree: Pivoting Metric Tree for Similarity Search in Multimedia Databases. ADBIS, Computer and Automation Research Institute Hungarian Academy of Science (2004)
20. Skopal, T., Pokorný, J., Snášel, V.: Nearest neighbours search using the PM-Tree. In: Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA 2005), Beijing, China, April 17-20, 2005. Lecture Notes in Computer Science, vol. 3453, pp. 803–815. Springer (2005)
21. Smith, J.R.: MPEG7 Standard for Multimedia Databases. SIGMOD Record (2001). https://doi.org/10.1145/376284.375814
22. Traina, C., Traina, A., Faloutsos, C., Seeger, B.: Fast indexing and visualization of metric data sets using Slim-trees. IEEE Transactions on Knowledge and Data Engineering (2002). https://doi.org/10.1109/69.991715
23. Uhlmann, J.K.: Satisfying general proximity/similarity queries with metric trees. Information Processing Letters **40**(4), 175–179 (1991)
24. Vilar, J.M.: Reducing the overhead of the AESA metric-space nearest neighbour searching algorithm. Information Processing Letters **56**(5), 265–271 (1995)
25. Zhou, X., Wang, G., Yu, J.X., Yu, G.: M+-tree: a new dynamical multidimensional index for metric spaces. In: Proceedings of the 14th Australasian Database Conference. pp. 161–168 (2003)
26. Zhou, X., Wang, G., Zhou, X., Yu, G.: BM+-tree: A hyperplane-based index method for high-dimensional metric spaces. In: Lecture Notes in Computer Science. pp. 398–409 (2005). https://doi.org/10.1007/11408079_36