Pattern Recognition Letters
journal homepage: www.elsevier.com

An efficient tree structure for indexing feature vectors

The-Anh Pham[a,b,**], Sabine Barrat[a], Mathieu Delalandre[a], Jean-Yves Ramel[a]

[a] *Laboratory of Computer Science, Francois Rabelais University, Tours 37200, France.*
[b] *Hong Duc University, Thanh Hoa city, Vietnam.*

ARTICLE INFO

Article history:

Communicated by S. Sarkar

Keywords:
Approximate Nearest neighbour Search
Feature Indexing
Randomized KD-trees
Priority Search

ABSTRACT

This paper addresses the problem of feature indexing in feature vector space. A linked-node m-ary tree (LM-tree) structure is presented to quickly produce the queries for an approximate and exact nearest neighbour search. Three main contributions are made, which can be attributed to the proposed LM-tree. First, a new polar-space-based method of data decomposition is presented to construct the LM-tree. Second, a novel pruning rule is proposed to efficiently narrow down the search space. Finally, a bandwidth search method is introduced to explore the nodes of the LM-tree. Extensive experiments were performed to study the behaviour of the proposed indexing algorithm. These experimental results showed that the proposed algorithm provides a significant improvement in the search performance for the tasks of both *exact* and *approximate* nearest neighbour (ENN/ANN) searches.

© 2014 Elsevier Ltd. All rights reserved.

1. Introduction

Fast nearest neighbour search is of central importance for many computer vision systems, such as systems for object matching, object recognition, and image retrieval. Although a large number of indexing algorithms have been proposed in the literature, few of them (e.g., LHS-based schemes of Lv et al. (2007), randomized KD-trees of Silpa-Anan and Hartley (2008), randomized K-medoids clustering trees of Muja and Lowe (2012), and the hierarchical K-means tree of Muja and Lowe (2009)) have been well validated with extensive experiments to show satisfactory performance on specific benchmarks.For a comprehensive survey of the indexing algorithms in vector space, we refer to the work of Böhm et al. (2001). These algorithms are often categorized into space-partitioning, clustering, hashing and hybird approaches. We will discuss hereafter the most representative methods.

Space-partitioning approach: Friedman et al. (1977) introduced KD-tree, whose basic idea is to iteratively partition the data X into two roughly equal-sized subsets, by using a hyperplane that is perpendicular to a split axis in a D-dimensional vector space. Searching for a nearest neighbour of a given query point q is accomplished by using a branch-and-bound technique. Several variations of the KD-tree have been investigated to address the ANN search. *Best-Bin-First* (BBF) search or priority search of Beis and Lowe (1997) is a typical improvement on the KD-tree. Its basic idea is twofold. First, it limits the maximum number of data points to be searched. Second, it visits the nodes in the order of increasing distances to the query. The use of priority search was further improved in Silpa-Anan and Hartley (2008). In this paper, the author constructed multiple *randomized* KD-trees (*RKD*-trees), each of which is built by selecting, randomly, at each node, a split axis from a few dimensions having the highest variance. A slight difference in the *RKD*-trees was also investigated in this work, where the data are initially aligned to the principal axes. Hence, the obtained indexing scheme is called principal component KD-trees (*PKD*-trees). Experimental results show significantly outstanding performance compared to the use of a single KD-tree. A last noticeable improvement in the KD-tree for the ENN search is the principal axis tree (PAT-tree) of McNames (2001). In the PAT-tree, the split axis is chosen as the principal axis, which has the highest variance in the underlying data. Therefore, the obtained regions are treated as hyper-polygons rather than as

**Corresponding author: Tel.: +84-912-721-200;
e-mail: phamtheanh@hdu.edu.com (The-Anh Pham)

hyper-rectangles, as in the KD-tree. Consequently, this approach complicates the process of computing the lower bound of the distance from the query to a given node.

Clustering approach: The clustering-based indexing methods differ from the space-partitioning-based methods mainly in the step of tree construction. Instead of dividing the data by using a hyper-plane, these methods employ a clustering method (e.g., K-means by Fukunaga and Narendra (1975), K-medoids by Muja and Lowe (2012)) to iteratively partition the underlying data into sub-clusters. The partitioning process is then repeated until the size of all of the sub-clusters falls below a threshold. Muja and Lowe (2009) extended the work in Fukunaga and Narendra (1975) by incorporating the use of priority queue to the hierarchical clustering tree. In their work, an ANN search proceeds by traversing down the tree and always chooses the node whose cluster centre is closest to the query. Each time that a node is selected for further exploration, the other sibling nodes are inserted into a priority queue that contains a sequence of nodes that are stored in increasing order of their distances to the query. This process continues when reaching a leaf node and is followed by a sequence search for the points contained in that node. Backtracking is then invoked, starting from the top node in the priority queue. During the search process, the algorithm maintains adding new candidate nodes to the priority queue. Experimental results show that the proposed algorithm gives better results than two other state-of-the-art methods. The hierarchical clustering tree in Muja and Lowe (2009) has been further extended in Muja and Lowe (2012) to build up multiple hierarchical clustering trees. The ANN search proceeds in parallel among the hierarchical clustering trees. Experimental results show that a significant improvement in the search performance is achieved and that the proposed algorithm can scale well to large-scale datasets.

Hashing approach: Locality-Sensitive Hashing (LHS) of Indyk and Motwani (1998) has been known as one of the most popular hashing-based methods. The key idea is to project the data into a large number of random lines in such a way that the likelihood of hashing two data points into the same bucket is proportional to their similarity degree. Given a query, a proximity search is proceeded by first projecting the query using the LSH functions. The obtained indices are then used to access the appropriate buckets followed by a sequence search for the data points contained in the buckets. Given a sufficiently large number of hash tables, the LSH can perform an ANN search in sublinear time complexity. Kulis and Grauman (2009) extended the LSH to the case in which the similarity function is an arbitrary kernel function κ: $D(p,q) = \kappa(p,q) = \phi(p)^T\phi(q)$, where $\phi(x)$ is an unknown embedding function. The main drawback of these two studies is the use of a very large memory space to construct the hash tables. To address this issue, Panigrahy (2006) introduced an entropy-based LSH indexing technique. Its basic idea is quite interesting: given a query q and a parameter r of the distance from q to its nearest neighbour, the *synthetic* neighbours of q within a distance r are randomly generated. These synthetic neighbours are then hashed by using the LSH functions. The obtained hash keys are used to access the bucket candidates, where it is expected that the true nearest neighbour of q could be present. A detailed analysis of the entropy-based LSH algorithm was reported by Lv et al. (2007); this paper showed that the entropy-based LSH algorithm does not give a noticeable search improvement compared to the original LSH method. Lv et al. (2007) proposed another approach, which is known as *multi-probe* LSH, to reduce the utilized hash tables. Its basic idea is to search multiple buckets, which probably contain the potential nearest neighbours of the query. The rationale is easily seen: if a data point p is close to another data point q but they are not hashed into the same bucket, then there is a high chance that they are hashed into two "close" buckets. Experimental results show a significant improvement in terms of the space efficiency, compared to the original LSH scheme. Recently, Aiger et al. (2013) introduced a variation of the LSH scheme based on random grids. A dataset X, which consists of n D-dimensional vectors, is randomly rotated and shifted up to $e^{D/c}$ times where c is an approximate factor of search precision (e.g., $c = 2$ in their experiments). For each rotation/translation, the corresponding dataset is partitioned using an uniform grid of cell size $w = \frac{c}{\sqrt{D}}$ and the points contained in each cell are hashed into the same bucket. Consequently, space overhead is a big concern of this method (i.e., $O(De^{D/c}n)$). For instance, if 128-dimensional SIFT features are used, the proposed method consumes a huge memory space of $O(128e^{64}n)$.

Hybird approach: Recent interests in ANN search have been moved towards product quantization which can be considered as a hybird fashion of space-partitioning and clustering approaches. The curial benefit of using PQ for ANN search is the capability of indexing extremely large-scale and high-dimensional datasets. The essence of Product Quantization (PQ) in the work of Jégou et al. (2011) is to uniformly decompose the original data space into distinct sub-spaces and then to create separately an optimized sub-codebook for the data in each sub-space using K-means algorithm. Non exhaustive search is proceeded by employing an inverted file structure. However, as PQ employs an axis-aligned grid for space decomposition, many centroids are created without the support of data. This damages the search performance because it has been commonly agreed that better fitting to the underlying data is crucial for achieving good search speed and search accuracy. Ge et al. (2014) extends PQ by introducing Optimized Product Quantization (OPQ). The main spirit of OPQ is to design a quantizer that solves the quantization optimization problem in both aspects of space decomposition and sub-codebook construction. Doing so, OPQ allows the split grid to be rotated by arbitrary orientation to make better fitting of the underlying distribution. Although OPQ significantly outperforms PQ in search performance, such an alignment is less-efficient to the cases of multi-model distribution as discussed by Kalantidis and Avrithis (2014). To address this issue, Kalantidis and Avrithis (2014) proposed optimizing locally the PQ per centroid for quantizing the residual distribution. However, learning locally an optimized PQ is not an efficient process if a non-parametric optimization fashion is used. Alternatively, a parametric learning technique is often employed which requires an assumption of some prior knowledge about the underlying distribution (e.g., typically a Gaussian distribution). Consequently,

search performance would be significantly impacted if the assumption is not satisfied.

In this work, we conduct a detailed investigation of our previous work in Pham et al. (2013) for indexing the feature vectors. Specifically, we have carried out the following extensions:

- We provide a deeper and wider review of related work by including recent progress for ANN search (e.g., product quantization approaches).
- The spirit of the proposed approach is presented in great details with deeper analysis and illustration.
- Additional experiments (e.g., more datasets) and an application to image retrieval are included for better evaluation of the proposed approach.
- A thorough study of parameter impact is performed coupling with the addition of an automatic parameter tuning algorithm to make the proposed indexing scheme well-adapted to a specific dataset and search precision.

For the remainder of this paper, Section 2 presents the proposed approach. Section 3 dedicates to performance evaluation. Section 4 concludes the paper and defines some future work.

2. The LM-tree indexing algorithm

2.1. Construction of the LM-tree

For a better presentation of our approach, we use the notation $\mathbf{p}$ as a point in the R^D feature vector space, and p_i as the i^{th} component of $\mathbf{p}$ ($1 \leq i \leq D$). We also denote $p = (p_{i_1}, p_{i_2})$ as a point in a $2D$ space. We adopted here the conclusion made in Silpa-Anan and Hartley (2008) about the use of PCA for aligning the data before constructing the LM-tree. This approach enables us to partition the data via the narrowest directions. In particular, the dataset X is translated to its centroid following a step of data rotation to make the coordinate axes aligned with the principal axes. Note that no dimension reduction is performed in this step. In fact, PCA analysis is used only to align the data. Next, the LM-tree is constructed by recursively partitioning the dataset X into m roughly equal-sized subsets as follows:

- Sort the axes in decreasing order of variance, and choose randomly two axes, i_1 and i_2, from the first L highest variance axes ($L < D$).
- Project every point $\mathbf{p} \in X$ into the plane $i_1\mathbf{c}i_2$, where $\mathbf{c}$ is the centroid of the set X, and then compute the corresponding angle: $\phi = \arctan(p_{i_1} - c_{i_1}, p_{i_2} - c_{i_2})$.
- Sort the angles $\{\phi_t\}_{t=1}^{n}$ in increasing order ($n = |X|$), and then divide the angles into m disjointed sub-partitions: $(0, \phi_{t_1}] \cup (\phi_{t_1}, \phi_{t_2}] \cup \ldots \cup (\phi_{t_m}, 360]$, each of which contains roughly the same number of elements (e.g., the data points projected into the plane $i_1\mathbf{c}i_2$).
- Partition the set X into m subsets $\{X_k\}_{k=1}^{m}$ corresponding to m angle sub-partitions obtained in the previous step.

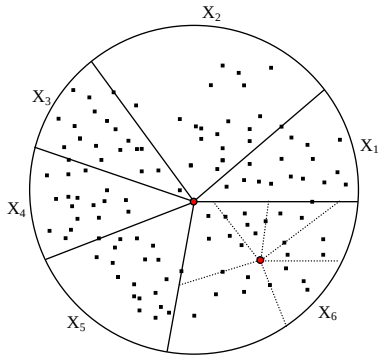


Fig. 1. Illustration of the iterative process of data partitioning in a 2D space: the 1^{st} partition is applied to the dataset X, and the 2^{nd} partition is applied to the subset X_6 (the branching factor $m = 6$).

Algorithm 1 LMTreeBuilding(X, m, L, L_{max})

$A \leftarrow$ sort the axes in the decreasing order of variance
$t_1 \leftarrow random(L)$ {select randomly a number in $[1, L]$}
$t_2 \leftarrow random(L)$
Ensure: $t_1 \neq t_2$
$i_1 \leftarrow A[t_1]$, $i_2 \leftarrow A[t_2]$ {select two axes i_1 and i_2}
$\mathbf{c} \leftarrow$ compute the centroid of X
$i \leftarrow 1$, $n \leftarrow |X|$
while $i \leq n$ **do**
  $\mathbf{p} \leftarrow X[i]$
  $\phi_i = \arctan(p_{i_1} - c_{i_1}, p_{i_2} - c_{i_2})$
  $i \leftarrow i + 1$
end while
$B \leftarrow$ sort the angles ϕ_i in the increasing order
$\Delta \leftarrow n/m$
$X_k \leftarrow \emptyset$ {reset each subset X_k to empty ($1 \leq k \leq m$)}
for each $\mathbf{p} \in X$ **do**
  $\phi_{\mathbf{p}} = \arctan(p_{i_1} - c_{i_1}, p_{i_2} - c_{i_2})$
  $k \leftarrow 1$
  while $k \leq m$ **do**
    if $\phi_{\mathbf{p}} \leq B[k\Delta]$ **then**
      $X_k \leftarrow X_k \cup \{p\}$ {put $\mathbf{p}$ into a proper subset X_k}
      $k \leftarrow m + 1$ {break the loop}
    end if
    $k \leftarrow k + 1$
  end while
end for
$k \leftarrow 1$
while $k \leq m$ **do**
  $T_k \leftarrow$ construct a new node corresponding to X_k
  if $—X_k| < L_{max}$ **then**
    LMTreeBuilding(X_k, m, L, L_{max}) {recursive tree building for each subset X_k}
  end if
end while

For each subset X_k, a new node T_k is constructed and then attached to its parent node, where we also store the following information: the split axes (i.e., i_1 and i_2) and the split centroid (c_{i_1}, c_{i_2}), the split angles $\{\phi_{t_k}\}_{k=1}^{m}$, and the split projected points $\{(p_{i_1}^k, p_{i_2}^k)\}_{k=1}^{m}$, where the point $(p_{i_1}^k, p_{i_2}^k)$ corresponds to the split angle ϕ_{t_k}. For efficient access across these child nodes, a di-

rect link is established between two adjacent nodes T_k and T_{k+1} ($1 \leq k < m$), and the last one T_m is linked to the first one T_1. Next, we repeat this partitioning process for each subset X_k that is associated with the child node T_k until the number of data points in each node falls below a pre-defined threshold L_{max}. Figure 1 illustrates the first and second levels of the LM-tree construction with a branching factor of $m = 6$ and the whole process is illustrated in Algorithm 1.

It is worthwhile pointing out that each time that a partition proceeds, two axes are employed for dividing the data. This approach is in contrast to many existing tree-based indexing algorithms, where only one axis is employed to partition the data. Consequently, as argued in Silpa-Anan and Hartley (2008), considering a high-dimensional feature space, such as 128-dimensional SIFT features, the total number of axes involved in the tree construction is rather limited, making any pruning rules inefficient, and the tree is less discriminative for later usage of searching. Naturally, the number of principal axes involved in partitioning the data is proportional to both the search efficiency and precision.

2.2. *Exact nearest neighbour search in the LM-tree*

ENN search in the LM-tree is proceeded by using a branch-and-bound algorithm. Given a query point $\mathbf{q}$, we first project $\mathbf{q}$ into a new space by using the principal axes, which is similar to how we processed the LM-tree construction. Next, starting from the root, we traverse down the tree, and we use the split information stored at each node to choose the best child node for further exploration. Specifically, given an internal node u along with the corresponding split information $\{i_1, i_2, c_{i_1}, c_{i_2}, \{\phi_{t_k}\}_{k=1}^m, \{(p_{i_1}^k, p_{i_2}^k)\}_{k=1}^m\}$ which is already stored at u, we first compute an angle: $\phi_{q_u} = \arctan(q_{i_1} - c_{i_1}, q_{i_2} - c_{i_2})$. Next, a binary search is applied to the query angle ϕ_{q_u} over the sequence $\{\phi_{t_k}\}_{k=1}^m$ to choose the child node of u that is closest to the query $\mathbf{q}$ for further exploration. This process continues until a leaf node is reached, followed by the partial distance search (PDS) of De-Yuan Cheng (1984) to the points contained in the leaf. Backtracking is then invoked to explore the remainder of the tree. Algorithm 2 illustrates the main steps of this process.

Each time, when we are positioned at a node u, the lower bound is computed as the distance from the query $\mathbf{q}$ to the node u. If the lower bound exceeds the distance from $\mathbf{q}$ to the nearest point found so far, we can safely avoid exploring this node and proceed with other nodes. In this section, we present a novel rule for computing efficiently the lower bound. Our pruning rule was inspired by the Principal Axis Tree (PAT) of McNames (2001). The disadvantages of the PAT rule are the computational cost of the complexity (i.e., $O(D)$) and the inefficiency when working on a high-dimensional space because only one axis is employed at each partition. Because our method of data decomposition (i.e., LM-tree construction) is quite different from that of the KD-tree-based structures, we have developed a significant improvement of the pruning rule used in PAT.

It is worth mentioning that the nearest neighbors found in the PCA space are identical to those found in the original space. Specifically, in our work PCA is used not to reduce the data dimension, but to make better fitting of the underlying data by applying two basic transformations of translation and rotation. Here, the dataset is pre-processed to be zero-centered and is rotated to make the coordinate axes aligned with the principal axes. As both the query and database are undergone by such uniform transformations, the distances are preserved in the PCA space. More precisely, all the distances are uniformly scaled by the same constant.

Algorithm 2 ENNSearch(u, $\mathbf{q}$, $dist_{lb}$)

Input: A pointer to a current node of the LM-tree (u), a query ($\mathbf{q}$) and the current lower bound distance ($dist_{lb}$)
Output: The exact nearest neighbor of $\mathbf{q}$
if u is a leaf node **then**
  $\{\mathbf{p}_{best}, dist_{best}\} \leftarrow$ sequence search for the points at u
else
  $\{i_1, i_2, c_1, c_2, \{\phi_{t_k}\}_{k=1}^m\} \leftarrow$ access split information at u
  $\phi_{q_u} = \arctan(q_{i_1} - c_1, q_{i_2} - c_2)$
  $s_k \leftarrow$ find a son of u where ϕ_{q_u} is contained in $(\phi_{t_{k-1}}, \phi_{t_k}]$
  ENNSearch(s_k, $\mathbf{q}$, $dist_{lb}$) {explore s_k first}
  $m \leftarrow$ the number of sons of u
  $L_{ord} \leftarrow \emptyset$ {construct an ordered list of visiting the nodes}
  $s_{left} \leftarrow s_k$, $s_{right} \leftarrow s_k$, $i \leftarrow 1$
  while $i \leq m/2$ **do**
    $L_{ord} \leftarrow L_{ord} \cup move2left(s_{left})$
    $L_{ord} \leftarrow L_{ord} \cup move2right(s_{right})$ {collect adjacent nodes in increasing order from s_k}
    $i \leftarrow i + 1$
  end while
  for each node s in L_{ord} **do**
    $\{dist_{nlb}, \mathbf{q}_n\} \leftarrow$ ComputeLB(s, $\mathbf{q}$, $dist_{lb}$) {update new lower bound and query}
    if $dist_{nlb} < dist_{best}$ **then**
      ENNSearch(s, $\mathbf{q}_n$, $dist_{nlb}$)
    end if
  end for
end if
return $\mathbf{p}_{best}$ {the exact nearest neighbor of $\mathbf{q}$}

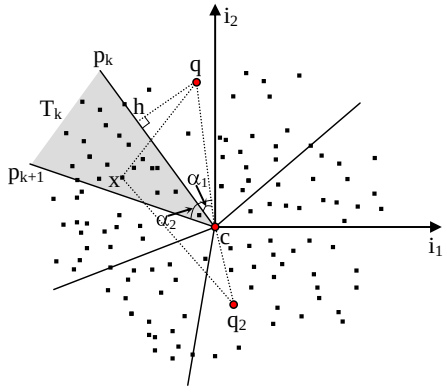


Fig. 2. Illustration of the lower bound computation.

Let u be the node in the LM-tree at which we are positioned, let T_k be one of the children of u that is going to be searched, and let $p_k = (p_{i_1}^k, p_{i_2}^k)$ be the k^{th} split point, which corresponds to the child node T_k (Figure 2). The lower bound $LB(\mathbf{q}, T_k)$, from

$\mathbf{q}$ to T_k, is recursively computed from $LB(\mathbf{q}, u)$. This process is summarized as follows in correspondence to Algorithm 3:

- Compute the angles: $\alpha_1 = \angle qcp_k$ and $\alpha_2 = \angle qcp_{k+1}$, where $q = (q_{i_1}, q_{i_2})$ and $c = (c_{i_1}, c_{i_2})$.
- If at least one of the angles is smaller than 90^0 (e.g., the point q in Figure 2), then we have the following fact due to the rule of cosines in McNames (2001):

$$d(q, x)^2 \geq d(q, h)^2 + d(h, x)^2 \tag{1}$$

  where x is any point in the region of T_k, and $h = (h_{i_1}, h_{i_2})$ is the projection of q on the line cp_k or cp_{k+1}, while accounting for $\alpha_1 \leq \alpha_2$ or $\alpha_1 > \alpha_2$. Then, we applied the rule of lower bound computation in PAT in a 2D space as follows:

$$LB^2(\mathbf{q}, T_k) \leftarrow LB^2(\mathbf{q}, u) + d(q, h)^2 \tag{2}$$

  The point $\mathbf{h} = (q_1, q_2, \ldots, h_{i_1}, \ldots, h_{i_2}, \ldots, q_{D-1}, q_D)$ is then treated as a new query point in place of $\mathbf{q}$ in the means of lower bound computation to the descendant of T_k.
- If both angles, α_1 and α_2, are higher than 90^0 (e.g., the point q_2 in Figure 2), we have a more restricted rule:

$$d(q, x)^2 \geq d(q, c)^2 + d(c, x)^2 \tag{3}$$

  Therefore, the lower bound is easily computed as:

$$LB^2(\mathbf{q}, T_k) \leftarrow LB^2(\mathbf{q}, u) + d(q, c)^2 \tag{4}$$

  Again, the point $\mathbf{c} = (q_1, q_2, \ldots, c_{i_1}, \ldots, c_{i_2}, \ldots, q_{D-1}, q_D)$ is handled as a new query point in place of $\mathbf{q}$ in the means of lower bound computation to the descendant of T_k.

Because the lower bound $LB(\mathbf{q}, T_k)$ is recursively computed from $LB(\mathbf{q}, u)$, an initial value must be set for the lower bound at the root node. Obviously, we set $LB(\mathbf{q}, root) = 0$. It is also noted that when the point $\mathbf{q}$ is fully contained in the region of T_k, no computation of the lower bound is required.

Algorithm 3 ComputeLB(u, $\mathbf{q}$, $dist_{lb}$)

Input: A pointer to a node of the LM-tree (u), a query ($\mathbf{q}$) and the current lower bound distance ($dist_{lb}$)
Output: New lower bound ($dist_{nlb}$) and new query ($\mathbf{q}_n$)
$p_u \leftarrow$ get the parent node of u
$\{i_1, i_2, c_1, c_2\} \leftarrow$ access split information stored at p_u
$\alpha_{min} \leftarrow \min\{\alpha_1, \alpha_2\}$
$\mathbf{q}_n \leftarrow \mathbf{q}$
if $\alpha_{min} \leq 90^0$ **then**
    $\mathbf{h} \leftarrow$ projection of $\mathbf{q}$ on the split axis
    $\mathbf{q}_n[i_1] \leftarrow h_{i_1}$ {update the query at the positions i_1 and i_2}
    $\mathbf{q}_n[i_2] \leftarrow h_{i_2}$
    $dist_{nlb} \leftarrow dist_{lb} + (q_{i_1} - h_{i_1})^2 + (q_{i_2} - h_{i_2})^2$
else
    $\mathbf{q}_n[i_1] \leftarrow c_{i_1}$
    $\mathbf{q}_n[i_2] \leftarrow c_{i_2}$
    $dist_{nlb} \leftarrow dist_{lb} + (q_{i_1} - c_{i_1})^2 + (q_{i_2} - c_{i_2})^2$
end if
return $\{dist_{nlb}, \mathbf{q}_n\}$

2.3. Approximate nearest neighbour search in the LM-tree

The use of multiple randomized trees for ANN search has been incorporated with the priority search and successfully used in many tree-based structures by Muja and Lowe (2009) and Muja and Lowe (2012). Although the priority search was shown to give better search performance, it is subjected to a high computational cost because the process of maintaining a priority queue during the online search is rather expensive.

Here, we exploit the advantages of using multiple randomized LM-trees but without using the priority queue. The basic idea is to restrict the search space to the branches that are not very far from the considering path. In this way, we introduce a specific search procedure, a so-called *bandwidth* search, which proceeds by setting a search bandwidth to every intermediate node of the ongoing path. Particularly, let $P = \{u_1, u_2, \ldots, u_r\}$ be a considering path that is obtained by traversing a single LM-tree, where u_1 is the root node and u_r is the node at which we are positioned. The proposed bandwidth search indicates that for each intermediate node u_i of P ($1 \leq i \leq r$), every sibling node of u_i at a distance of $b + 1$ nodes ($1 \leq b < m/2$) on both sides from u_i does not need to be searched. The value b is called search bandwidth.

There is a notable point that when the projected query q is too close to the projected centroid c, all of the sibling nodes of u_i should be inspected as in the case of an ENN search. Specifically, this scenario occurs at a certain node u_i if $d(q, c) \leq \epsilon D_{med}$, where D_{med} is the median value of the distances between c and all of the projected data points that are associated with u_i, and ϵ is a tolerance radius parameter. In addition, to obtain a varying range of the search precision, we would need a parameter E_{max} for the maximum data points to be searched on a single LM-tree. As we are designing an efficient solution dedicated to an ANN search, it would make sense to use an approximate pruning rule rather than an exact rule. Specifically, we have used only formula (4) as an approximate pruning rule. This adaption offers two favourable features. First, it reduces much of the computational cost. Recall that rule (4) requires the computation of $d(q, c)$ in a 2D space, where point c has been already computed during the offline tree construction. The computation of this rule is thus very efficient. Second, it also ensures that a larger fraction of nodes will be inspected but few of them would be actually searched after checking the lower bound. In this way, it increases the chance of reaching the true nodes that are closest to the query. More generally, we have adapted formula (4) as follows:

$$LB^2(\mathbf{q}, T_k) \leftarrow \kappa \cdot (LB^2(\mathbf{q}, u) + d(q, c)^2) \tag{5}$$

where $\kappa \geq 1$ is the pruning factor that controls the rate of pruning the branches in the trees. This factor can be adaptively estimated during the tree construction given a specific precision and a specific dataset. However, we have set this value as $\kappa = 2.5$ and we show, in our experiments, that it is possible to achieve satisfactory search performance on many datasets. Algorithm 4 provides an outline to compute the lower bounds used in ANN search.

Algorithm 4 ComputeLB_ANN(u, $\mathbf{q}$, $dist_{lb}$, κ)

1: **Input:** A pointer to a node of the LM-tree (u), a query ($\mathbf{q}$), the current lower bound distance ($dist_{lb}$) and the pruning factor (κ)
2: **Output:** New (approximate) lower bound ($dist_{nlb}$) and new query point ($\mathbf{q}_n$)
3: $p_u \leftarrow$ get the parent node of u
4: $\{i_1, i_2, c_1, c_2\} \leftarrow$ access split information stored at p_u
5: $\mathbf{q}_n \leftarrow \mathbf{q}$
6: $\mathbf{q}_n[i_1] \leftarrow c_{i_1}$ {update the query at the positions i_1 and i_2}
7: $\mathbf{q}_n[i_2] \leftarrow c_{i_2}$
8: $dist_{nlb} \leftarrow dist_{lb} + \kappa \cdot ((q_{i_1} - c_{i_1})^2 + (q_{i_2} - c_{i_2})^2)$
9: **return** $\{dist_{nlb}, \mathbf{q}_n\}$

3. Experimental results

We have used four datasets for evaluating search performance of the proposed LM-tree algorithm. The first dataset SURF1M consists of one million 128-dimensional SURF features[1] that are extracted from the Zurich city building pictures[2]. Two datasets, ANN_SIFT1M and ANN_GIST1M, are well-known for ANN search performance evaluation in the literature that are provided by Jégou et al. (2011). The ANN_SIFT1M dataset contains one million 128-dimensional SIFT features whereas the ANN_GIST1M dataset is composed of one million 960-dimensional GIST features. Because the dimensionality of the GIST feature is very high and our computer configuration is limited (i.e., Windows XP, 2.4G RAM, Intel Core i5 1.7 GHz), we were not able to load the full ANN_GIST1M dataset into memory. Consequently, we have used 250000 GIST features for search evaluation. In addition, we have used a wide corpus set[3] of historical books that contain ornamental graphics to develop an application to image retrieval.

We evaluated our system versus several representative indexing systems in the literature, including randomized KD-trees (RKD-trees) of Silpa-Anan and Hartley (2008) that was combined with the priority search from Muja and Lowe (2009), hierarchical K-means tree (K-means tree) from Muja and Lowe (2009), randomized K-medoids clustering trees (RC-trees) from Muja and Lowe (2012), and the multi-probe LSH algorithm from Lv et al. (2007). These indexing systems were well-implemented and widely used in the literature because of the open source FLANN library[4]. The source code of our system is also publicly available at this address[5]. Note that the partial distance search was implemented in these systems to improve the efficiency of sequence search at the leaf nodes.

Following the convention of the evaluation protocol used in the literature by Beis and Lowe (1997) and Muja and Lowe (2009), we computed the search precision and search time as the average measures obtained by running 1000 queries taken from separated test sets. To make the results independent on the machine and software configuration, the speedup factor is computed relative to the brute-force search. In all the following experiments, the speedup is computed as the ratio of the search time obtained by running the brute-force search to the search time obtained by running an indexing system. The processing time for constructing the indexing trees and/or parameter tuning is not taken into account of speedup calculation because all of these steps are done once in an offline phase.

3.1. ENN search evaluation

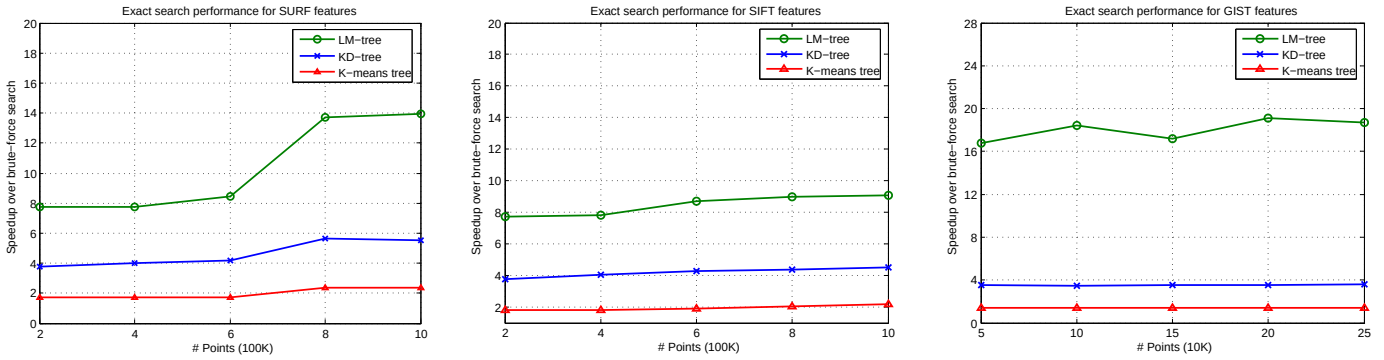
For ENN search, we use a single LM-tree and set the parameters involved in the LM-tree as follows: $L_{max} = 10$, $L = 2$, and $m \in \{6, 7, 7\}$ with respect to the GIST, SIFT and SURF features. By setting $L = 2$, we choose exactly the two highest variance axes at each level of the tree for data partitioning. We compared the performance of the ENN search of the following systems: the proposed LM-tree, the KD-tree and the hierarchical K-means tree. Figure 3 shows the speedup over the brute-force search of the three systems when applied to the SURF, SIFT and GIST features. Specifically, we compute the speedup of all the systems when varying the dataset size. As can be seen, for ENN search the two baseline indexing algorithms slightly achieve better search performance compared to the brute-force search when testing on all the three datasets. We can also clearly note that the proposed indexing LM-tree outperforms the other two systems on all of the tests. Taking the test where $\#Points = 150000$ on Figure 3(c), for example, the LM-tree gives a speedup of 17.2, the KD-tree gives a speedup of 3.53, and the K-means tree gives a speedup of 1.42 over the brute-force search. These results confirm the efficiency of the LM-tree for the ENN search relative to the two baseline systems.

3.2. ANN search evaluation

For the ANN search, we adopted here the benefit of using multiple randomized trees which were successfully used in the previous works of Silpa-Anan and Hartley (2008); and Muja and Lowe (2012). Hence, we set the required parameters as follows: $L_{max} = 10$, $L = 8$ and $b = 1$. By setting $L = 8$, we choose randomly two axes from the eight highest variance axes at each level of the tree for data partitioning. In addition, the parameter $b = 1$ indicates that our bandwidth search process visits three adjacent nodes (including the node in question) at each level of the LM-tree. Four systems participated in this evaluation, including the proposed LM-trees, RKD-trees, RC-trees, and K-means tree. We used 8 parallel trees in the first three systems, and we used a single tree for the K-means indexing algorithm because it was shown in Muja and Lowe (2009) that the use of multiple K-means trees does not give better search performance. For all of the systems, the parameters E_{max} and ϵ (i.e., the LM-tree) are varied to obtain a wide range of search precision.

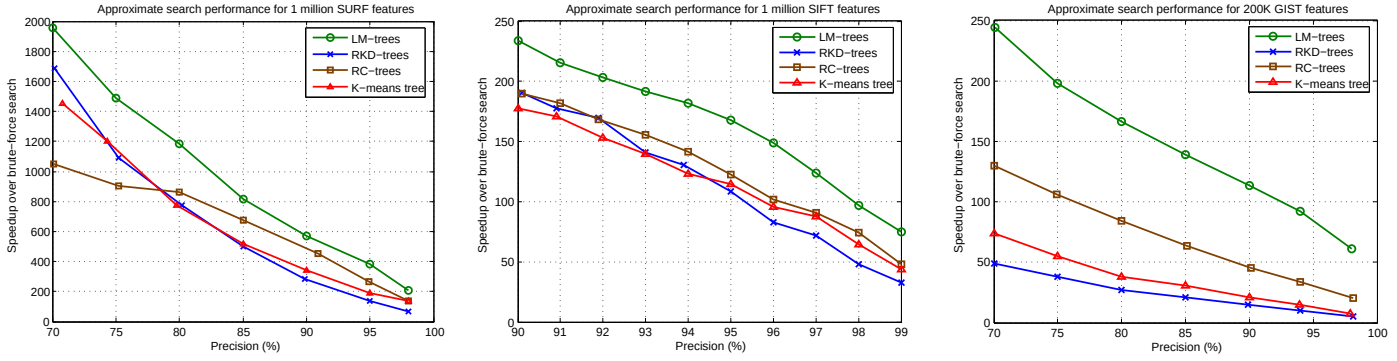
Figure 4(a) compares the search performance of four systems for one million SURF features. This figure shows that all the systems work rather efficiently for the SURF features and the proposed LM-tree algorithm markedly outperforms the other systems. For a reasonably high precision (e.g., 80%), the

[1] http://www.vision.ee.ethz.ch/ surf/index.html
[2] http://www.vision.ee.ethz.ch/showroom/zubud/index.en.html
[3] http://www.bvh.univ-tours.fr/
[4] http://www.cs.ubc.ca/ mariusm/index.php/FLANN/FLANN
[5] https://sites.google.com/site/ptalmtree/



(a) (b) (c)

Fig. 3. ENN search performance evaluation for the SURF (a), SIFT (b) and GIST (c) features of three indexing algorithms.



(a) (b) (c)

Fig. 4. ANN search performance evaluation for the SURF (a), SIFT (b) and GIST (c) features of four indexing algorithms.

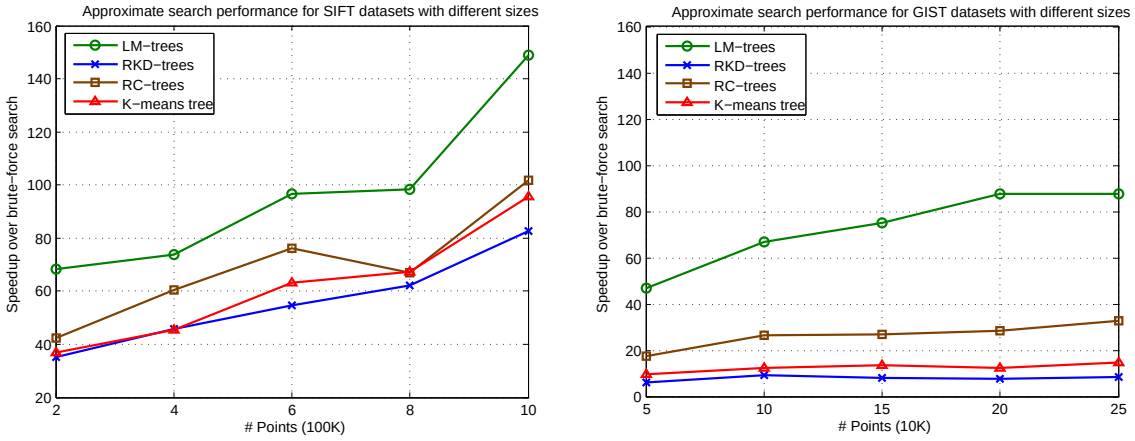
LM-tree algorithm achieves a speedup of more than three orders of magnitude (i.e., 1200 times) compared to the brute-force search while the best baseline algorithm (i.e., RC-trees) gives a speedup of around 860 times over the brute-force search.

Figure 4(b) shows the search speedup versus the search precision of the four systems for one million SIFT features. As can be seen, the proposed LM-trees algorithm gives significantly better search performance everywhere compared with the other systems. When considering the search precision of 95%, for example, the speedups over a brute-force search of the LM-trees, RKD-trees, RC-trees, and K-means tree are 167.7, 108.4, 122.4, and 114.5, respectively. To make it comparable with the multi-probe LSH indexing algorithm, we converted the real SIFT features to the binary vectors and attempted several parameter settings to obtain the best search performance. However, the result obtained on one million SIFT vectors is rather limited. Taking the search precision of 74.7%, for example, the speedup over a brute-force search (using Hamming distance) is only 1.5.

Figure 4(c) presents the search performance of all of the systems for 200000 GIST features. Again, the LM-trees algorithm clearly outperforms the others and tends to perform much better than the SIFT features. The RC-trees algorithm also works reasonably well, while the RKD-trees and K-means tree work poorly for this dataset. Considering the search precision of 90%, for example, the speedups over a brute-force search of the LM-trees, RKD-trees, RC-trees, and K-means tree are 113.5, 15.0, 45.2, and 21.2, respectively.

In Figure 5, we present the ANN search performance of the four systems as a function of the dataset size. For this purpose, the search precision is set to a rather high degree, especially at 96% and 95% for the SIFT and GIST features, respectively. This time, the LM-trees algorithm still gives a substantially better search performance than the others and tends to perform quite well, considering to the increase in the dataset size. For the SIFT features, the RC-trees algorithm works reasonably well, except for the point where $\#Points = 800K$ at which its search performance is noticeably degraded. It is also noted that the speedups of the RKD-trees and K-means tree for the GIST features are quite low, even lower than the speedup of the LM-tree for the ENN search.

Three crucial factors explain these outstanding results of the LM-trees. First, the use of the two highest variance axes for data partitioning in the LM-tree gives a more discriminative representation of the data in comparison to the common use of the sole highest variance axis as in the literature. Second, by using the approximate pruning rule, a larger fraction of nodes will be inspected, but many of them would be eliminated after checking the lower bound. In this way, the number of data points that will be actually searched is retained under the pre-



(a) (b)

Fig. 5. ANN search performance as a function of the dataset size: (a) SIFT dataset (search precision = 96%); (b) GIST dataset (search precision = 95%).

defined threshold E_{max}, while covering a larger number of inspected nodes and, thus, increasing the chance of reaching the true nodes that are closest to the query. Finally, the use of bandwidth search gives much benefit in terms of the computational cost, compared to the priority search that is used in the baseline indexing systems.

3.3. Complexity evaluation

In this section we provide further empirical evaluation of computational complexity of the proposed LM-tree algorithm in comparison with other indexing methods. For this purpose, we compute the absolute processing time for each indexing system for both ANN and ENN search problems when running on 200000 960-dimensional GIST features. Table 1 summarizes the results of each indexing method. Here, we denote t^b_{ENN} and t^b_{ANN} as the processing times for building the indexing tree(s) for ENN and ANN search, respectively. In addition, we also separate the processing time, t^b_{PCA}, for PCA computation that is used in the proposed LM-tree algorithm. It is worth mentioning that the tasks of PCA computation and tree building are done once in an offline phase. During the (online) search phase, we compute the mean search times t^s_{ENN} and t^s_{ANN} for ENN and ANN search, respectively, when running 1000 queries for each indexing method.

Table 1. Complexity report (seconds) of the LM-tree and other methods

Method	Tree building (offline)			Search (online)	
	t^b_{PCA}	t^b_{ENN}	t^b_{ANN}	t^s_{ENN}	t^s_{ANN}
RKD-tree(s)	0	7.4	55.6	0.114	0.046
RC-tree(s)	0	NA	242.3	NA	0.015
K-means tree	0	285.9	285.9	0.241	0.035
LM- tree(s)	156.8	7.1	24.7	0.023	0.004

Table 1 shows that the proposed LM-tree works very efficiently compared to other methods[6] for both ENN and ANN search. During the tree building phase, the proposed LM-tree takes only 7.1 and 24.7 seconds to construct the indexing tree(s) for ENN search (single tree) and ANN search (8 trees), respectively. However, as can be seen that the most consuming time step of LM-tree building is the computation of the principal components (i.e., 156.8 seconds), while for the three baseline methods this step is not taken into account because PCA is not used in these systems. During the search phase, the proposed LM-tree takes 0.023 and 0.004 (seconds) on average to fulfill a query for ENN and ANN search, respectively.

[6]The RC-tree(s) method is not used to deal with ENN search, so we marked this system as Not Applicable (NA).

3.4. Parameter tuning

In this section we study the effects of the parameters that are involved in the LM-tree on the search performance. Two types of parameters are involved in the LM-tree: static and dynamic parameters. The static parameters are those that are involved in the offline phase of building the LM-tree, including the maximum number of data points at a leaf node (L_{max}), the number of axes having the highest variance (L), and the branching factor (m). The dynamic parameters are used in the online phase of searching, including the search bandwidth (b), the maximum number of data points to be searched in a single LM-tree (E_{max}), the pruning factor ($\kappa = 2.5$), and the tolerance radius ϵ. For the static parameters, our investigation showed that they produce a negligible effect on the search performance given an appropriate setting of the dynamic parameters. Specifically, it was found that the following settings for the static parameters often lead to relatively good search performance: $L_{max} = 10$, $L = 8$, and $m \in \{6, 7\}$. The dynamic parameters (E_{max}, ϵ, b) are used to achieve a given specific search precision. They are thus treated as precision-driven parameters. This arrangement implies that given a specific precision P ($0\% \leq P \leq 100\%$), we can design a method to determine automatically the optimized settings for these parameters to achieve the best search time. In our case, we have set the parameter b to 1 and designed the following procedure to estimate the optimized settings for E_{max} and ϵ:

- Step 1: Sample the parameter ϵ into discrete values: $\{\epsilon_0, \epsilon_0 + \Delta, \ldots, \epsilon_0 + l\Delta\}$. In our experiments, we set: $\epsilon_0 = 0$, $\Delta = 0.04$, $l = 20$.

- Step 2: For each value $\epsilon_i = \epsilon_0 + i\Delta$ $(0 \leq i \leq l)$:
  - Step 2(a): Estimate an initial value for E_{max} by running the approximate search procedure without consideration of the parameter E_{max}. In this way, the search procedure terminates early with respect to the current settings of ϵ_i and b. Assume Q be the number of searched points during this process.
  - Step 2(b): Compute the precision P_i and speedup S_i by using the groundtruth information: if $P_i < P$, then proceed with a new value of ϵ_{i+1} and go to Step 2(a); otherwise, go to Step 2(c).
  - Step 2(c): Apply a binary search to find the optimized value of E_{max} in the range of $[0, Q]$. We first set $E_{max} = Q/2$ to run the approximate search procedure. Next, the search range is updated as either $[0, \frac{Q}{2}]$ or $[\frac{Q}{2}, Q]$ depending on whether $P_i > P$ or $P_i \leq P$. This process continues until the search range is unit-length.
  - Step 2(d): Update the best speedup, the parameter ϵ_i, and the optimized parameter E_{max} obtained from Step 2(c). Proceed with a new value of ϵ_{i+1} and go to Step 2(a) to find the better solution.
- Step 3: Return the parameters ϵ_i and E_{max}, corresponding to the best speedup found so far.

The rationale of setting the parameter b to 1 was inspired from the work of Lv et al. (2007). When using the two highest variance axes for partitioning the data, two close points can be divided into two adjacent bins. It is, thus, necessary to have a look at the adjacent bins while exploring the tree.

Figure 6 shows the search performance as a function of the pruning factor κ. In this experiment, we study the effect of κ for a wide range of search precision. For this purpose, the precision is set to 95%, 90%, and 80% for both the SIFT and GIST features. It can be seen that for both types of features, the speedup increases strongly with respect to the increase in κ to a certain extent. For example, taking the curve that corresponds to the precision $P = 90\%$ for the SIFT features, the speedup starts to decrease for $\kappa \geq 3.5$. Figure 6 also reveals that when increasing κ, we achieve much of the speedup for low precision ($P = 80\%$) compared with that for higher precision ($P \geq 90\%$). This relationship is referred to as the problem of over-pruning because a large number of branches have been eliminated as κ increases. Hence, the search process would miss some true answers. Fortunately, this matter can be resolved by using a cross-validation process, as suggested by Muja and Lowe (2009), during the tree construction, to adaptively select an appropriate value for κ, when provided a specific precision and dataset.

3.5. Application to image retrieval

In this section, an application to image retrieval is investigated while using the proposed LM-tree indexing scheme. For this purpose, we selected a wide corpus set of historical books that contain ornamental graphics. The dataset, called "Lettrine", is composed of 23062 isolated graphical images that were extracted from old documents.

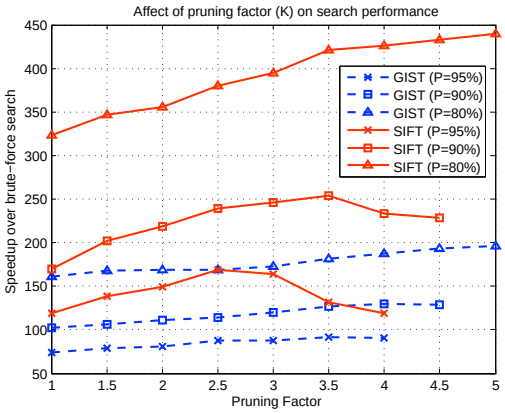


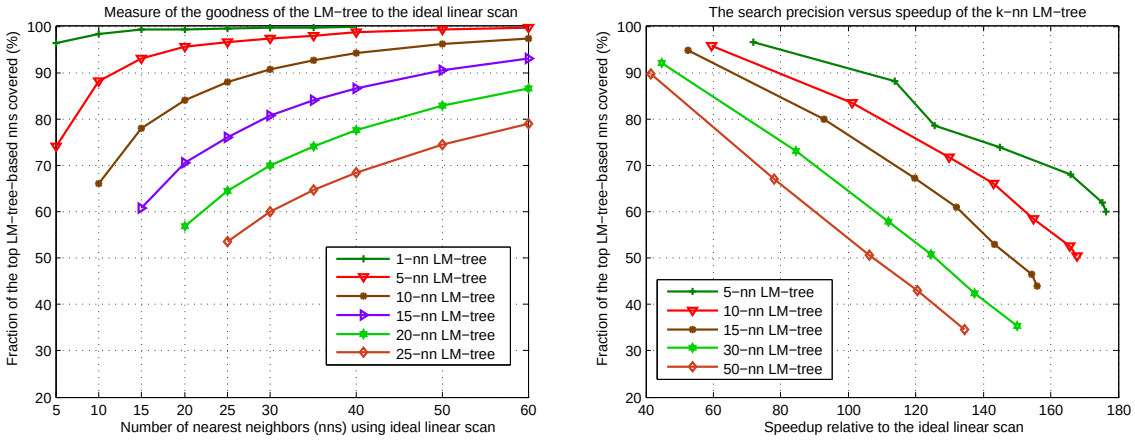
Fig. 6. Search performance as a function of the pruning factor: the precisions are set to 95%, 90%, and 80% for both the SIFT and GIST features.

Our challenge here is to prove that the proposed LM-tree indexing algorithm can be used in the context of a CBIR system. Hence, for the first step of feature extraction, we selected the GIST feature as it is a commonly used descriptor in the literature on image retrieval. We used the original implementation of the 512-dimensional GIST descriptor that was provided by the authors at this address[7]. Among 23062 ornamental graphics, 500 images were included in the query set and the remaining served as the database set. Next, the GIST features are computed once for all of the images in an offline phase. In the second step, all of the database GIST features are indexed by using our LM-tree algorithm.

We used the metric introduced by Kulis and Grauman (2009) to quantify the retrieval performance of our system. This metric measures how well a retrieval system can approximate an ideal linear scan. Specifically, we computed the fraction of the common answers, between our retrieval system and the ideal linear scan, to the number of answers of our system. Figure 7 (a) quantitatively shows how well our retrieval system approaches the ideal linear scan. These quantitative results are computed by using the top 1, 5, 10, 15, 20, and 25 ranked nearest neighbours of our system.

The results presented in Figure 7 (a) quantitatively show the quality of our retrieval system, which can be regarded as the search precision. Furthermore, we want to measure how fast the system is with respect to these results. For this purpose, Figure 7 (b) shows the speedup of our system relative to the ideal linear scan. In this test, both of the systems are evaluated by using the same parameter k for the number of nearest neighbours. With regard to the results presented in Figure 7 (b), we report in Table 2 the absolute search time (ms) and the fraction of searched points for the case of the 5-NNs LM-trees, which were averaged over the 500 queries. Taking a search precision of 73.9% for example, our system must explore 0.78% of the whole database and takes only 0.3 (ms) to return the top 5-NN answers.

[7] http://people.csail.mit.edu/torralba/code/spatialenvelope/



(a) (b)

Fig. 7. (a) Search quality of the k-NN LM-trees; (b) Search quality versus speedup of the k-NN LM-trees.

Table 2. Report of the mean search time (ms) and the fraction of searched points (%) for the 5-NN LM-trees.

Performance	Search precision (%)					
	61.8	68.1	73.9	78.6	88.1	96.5
Search fraction	0.32	0.52	0.78	1.17	2.45	6.21
Mean time	0.25	0.26	0.30	0.36	0.38	0.60

4. Conclusions and future work

In this paper, an efficient indexing algorithm in feature vector space has been presented. Three main features are attributed to the proposed LM-tree. First, a new polar-space-based method of data decomposition has been presented to construct the LM-tree. This new decomposition method differs from the existing methods in that the two highest variance axes of the underlying dataset are employed to iteratively partition the data. Second, a novel pruning rule is proposed to quickly eliminate the search paths that are unlikely to contain good candidates from the nearest neighbours. Furthermore, the lower bounds are easily computed, as if the data were in 2D space, regardless of how high the dimensionality is. Finally, a bandwidth search method is introduced to explore the nodes of the LM-tree. The proposed LM-tree has been validated on different datasets composed of SURF, SIFT and GIST features, which demonstrates that it works very well for both ENN and ANN searches compared to many state-of-the-art methods.

For further improvements to this work, dynamic insertion and deletion of the data points in the LM-tree would be incorporated to make the system adaptive to the dynamic changes in the data. The study of designing an indexing system, which can work on the data stored on an external disk, will be investigated to deal with extremely large datasets.

References

Aiger, D., Kokiopoulou, E., Rivlin, E., 2013. Random grids: Fast approximate nearest neighbors and range searching for image search, in: Proceedings of International Conference on Computer Vision (ICCV 2013), pp. 3471–3478.

Beis, J.S., Lowe, D.G., 1997. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces, in: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition, pp. 1000–1006.

Böhm, C., Berchtold, S., Keim, D.A., 2001. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. 33, 322–373.

De-Yuan Cheng, Gersho, A.R.B.S.Y., 1984. Fast search algorithms for vector quantization and pattern matching, in: IEEE International Conference on Acoustics, Speech, and Signal Processing, pp. 372–375.

Friedman, J.H., Bentley, J.L., Finkel, R.A., 1977. An algorithm for finding best matches in logarithmic expected time. ACM Trans. Math. Softw. 3, 209–226.

Fukunaga, K., Narendra, M., 1975. A branch and bound algorithm for computing k-nearest neighbors. IEEE Trans. Comput. 24, 750–753.

Ge, T., He, K., Ke, Q., Sun, J., 2014. Optimized product quantization. IEEE Trans. Pattern Anal. Mach. Intell. 36, 744–755.

Indyk, P., Motwani, R., 1998. Approximate nearest neighbors: towards removing the curse of dimensionality, in: Proceedings of the thirtieth annual ACM symposium on Theory of computing, pp. 604–613.

Jégou, H., Douze, M., Schmid, C., 2011. Product Quantization for Nearest Neighbor Search. IEEE Trans. Pattern Anal. Mach. Intell. 33, 117–128.

Kalantidis, Y., Avrithis, Y., 2014. Locally optimized product quantization for approximate nearest neighbor search, in: Proceedings of International Conference on Computer Vision and Pattern Recognition (CVPR 2014), Columbus, Ohio.

Kulis, B., Grauman, K., 2009. Kernelized locality-sensitive hashing for scalable image search, in: IEEE International Conference on Computer Vision, pp. 1–8.

Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K., 2007. Multi-probe lsh: efficient indexing for high-dimensional similarity search, in: Proceedings of the 33rd International Conference on Very large Databases, pp. 950–961.

McNames, J., 2001. A fast nearest-neighbor algorithm based on a principal axis search tree. IEEE Trans. Pattern Anal. Mach. Intell. 23, 964–976.

Muja, M., Lowe, D.G., 2009. Fast approximate nearest neighbors with automatic algorithm configuration, in: Proceedings of International Conference on Computer Vision Theory and Applications, pp. 331–340.

Muja, M., Lowe, D.G., 2012. Fast matching of binary features, in: Proceedings of the Ninth Conference on Computer and Robot Vision, pp. 404–410.

Panigrahy, R., 2006. Entropy based nearest neighbor search in high dimensions, in: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, pp. 1186–1195.

Pham, T.A., Barrat, S., Delalandre, M., Ramel, J.Y., 2013. An efficient indexing scheme based on linked-node m-ary tree structure, in: 17th International Conference on Image Analysis and Processing (ICIAP 2013), pp. 752–762.

Silpa-Anan, C., Hartley, R., 2008. Optimised kd-trees for fast image descriptor matching, in: IEEE Conference on Computer Vision and Pattern Recognition, pp. 1–8.