An efficient indexing scheme based on linked-node m-ary tree structure and polar partitioning of feature space

The-Anh Pham, Sabine Barrat, Mathieu Delalandre, and and Jean-Yves Ramel

Laboratoire d'Informatique 64, Avenue Jean Portalis, 37200 Tours - France.
the-anh.pham@etu.univ-tours.fr,{sabine.barrat, mathieu.delalandre, jean-yves.ramel}@univ-tours.fr

Abstract. Fast nearest neighbor (FNN) search is a crucial need of many recognition systems. Despite the fact that a large number of algorithms have been proposed in the literature for the FNN problem, few of them (e.g., randomized KD-trees, hierarchical K-means tree, randomized clustering trees, and LHS-based schemes) have been well validated on extensive experiments, and known to give satisfactory performance on specific benchmarks. While such representative indexing schemes works well for the approximate nearest neighbor search (ANN), their performance is slightly better or even worse than brute-force search for the problem of exact nearest neighbor search (ENN). In this work, we propose a linked-node m-ary tree (LM-tree) indexing algorithm, which works rather well for both the ANN and ENN tasks. The main contribution of the LM-tree is three-fold. 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 presented to deal with the ANN task, in combination with the use of multiple randomized LM-trees. Our experiments carried out on one million SIFT features show that the proposed method gives substantial improvement in search performance relative to the aforementioned indexing algorithms.

Keywords: Image Indexing, Nearest Neighbor Search, Hashing Function, Clustering Trees

1 Introduction

Recently, there has been a great interest of researchers to deal with the fast nearest neighbor search as this task plays a critical role in many computer vision systems such as object matching, object recognition, and CBIR. In many applications, an interested object can be represented by a real feature vector in a D-dimensional space. The problem of nearest neighbor search is formulated as follows. Given a set X composing of n points or feature vectors in R^D space, design a data structure for reorganization of X to efficiently answer the queries of finding a point in X closest to given any query point. Excellent surveys of indexing algorithms in vector space are presented by White et al. [16], Gaede et al. [6], and Böhm et al. [3]. These algorithms are often categorized into space-paritioning-based, clustering-based, and hashing-based approaches. We will discuss the most representative algorithms in the following sections.

For the space-partitioning-based approaches, KD-tree is probably argued as one of the most popular techniques [4]. The basic idea is to iteratively partition the data X into two roughly equal-sized subsets, using a hyperplane perpendicular to a split axis, say the i^{th} axis, in R^D space. The first subset contains the points whose values at the i^{th} dimension are smaller than a split value, and the second subset contains the rest of X. The split value is often chosen as the median value of the i^{th} components of the points in X. Two new nodes are then created with respect to the subsets. This process is then repeated for the two subsets until the size of every subset falls below a threshold. Searching for a nearest neighbor of given a query point q is proceeded using a branch-and-bound technique whose the pruning rule works as follows: a node u is selected to explore if its hyper-rectangle *does* intersect the hyper-sphere centered at q with a radius equal to the distance of q to the nearest neighbor found so far. The KD-tree has been shown to work very efficiently in low-dimensional space for the ENN search. Several variations of the KD-tree have been investigated to deal with the ANN search. The *Best-Bin-First* search or priority search in [1] is a typical improvement of the KD-tree. The basic idea of the BBF technique is twofold: it limits the maximum number of data points to be searched; and it visits the nodes in the order of increasing distances to the query. The BBF technique has been shown to give much better performance than restricted search with the KD-tree. The use of priority search is further improved in [15], where the author proposed to construct multiple KD-trees, called NKD-tree, by applying different rotations to the data. The obtained results are quite interesting. Based on that, the technique of using multiple KD-trees has been developed in two new different ways: multiple randomized KD-trees (RKD-trees) and multiple randomized principal component KD-trees (PKD-trees). The RKD-tree is constructed by selecting the split axis at random from a small set of dimensions having the highest variances. The PKD-tree is constructed in a similar manner but the data is aligned in advance to the principal axes obtained from PCA analysis. Experimental results show significantly better performance, compared to the original single KD-tree. A last noticeable improvement of the KD-tree for the ENN search is principle axis tree (PAT-tree) [11]. The PAT-tree extends the KD-tree at twofold. First, it constructs a bigger fanout tree by partitioning the data at each step into m subsets ($m \geq 2$). Second, the split hyper-plane is chosen to be perpendicular to the principle axis of data at each level of partition, resulting in the hyper-polygons of the nodes rather than the hyper-rectangles as in the KD-tree. Although the computation of lower bounds in the PAT-tree is more complicated, the PAT-tree still outperforms many other indexing schemes in the experiments.

For the clustering-based approaches, Fukunaga et al. [5] proposed to recursively partition the data into smaller regions using the K-means technique. This process terminates when the size of every region falls below a threshold, resulting in a hierarchical K-means tree of the data. Nearest neighbor searching is then proceeded by a branch-and-bound algorithm. Experiments reported that the proposed algorithm works quite efficiently. Muja and G. Lowe [12] extend the work of Fukunaga et al. by incorporating the priority search to deal with the ANN task. Particularly, proximity searching is proceeded by traversing down the tree and always choosing the node whose cluster center is closest to the query. Each time when we pick up a node for further exploration, the other sibling nodes are inserted to a priority queue, which contains a sequence of nodes stored in increasing order of the distance to the query. This continues until a leaf node is reached followed a sequence search for the points contained in this node. Backtracking is then invoked starting from the top node stored in the priority queue. The experiments shown a significant improvement compared to the LHS-based algorithms and the KD-tree for proximity searching on many datasets. Multiple randomized clustering trees have been also explored in [13] by the same authors, where the trees are constructed by selecting the centroids at random. In [14], Nister et al. constructed a hierarchical vocabulary tree for representation of feature vectors of MSER regions. The K-means algorithm is used to partition the feature vectors into smaller groups, where each is then associated with a node of tree containing the cluster center. This process is recursively repeated for each of the obtained groups until the height of the tree exceeds a pre-defined threshold. In this way, the tree is constructed and hierarchically defines the quantized cells of feature vectors, which could be regarded as the visual vocabularies. Proximity searching is essentially similar to other tree-based approaches by traversing down the tree, and at each level selecting the node whose center is closest to the query.

For hashing-based approaches, Locality-Sensitive Hashing (LHS) [7] has been known as one of the most popular hashing-based methods, which can perform ANN search with a truly sub-linear time even for very large-dimensional data. The key idea of LHS is to design the hash functions that the similar points are hashed with a high probability of collision, while the dissimilar points are likely to be hashed with different keys. Given a query, proximity searching is proceeded by first projecting the query using the LSH functions. The obtained indices are then used to access the appropriate buckets followed a sequence search for the data points contained in the buckets. Given a sufficiently large number of hash tables, the LSH can perform ANN search in a truly sub-linear time complexity. Qin Lv et al. [10] introduced *multi-probe* LSH to substantially reduce the number of hash tables, while retaining the same search precision. The basic idea is to look up multiple buckets probably containing the good candidates of nearest neighbors to the query. In this way, the proposed method reduces the space requirement and increases the chance of finding the true answers. Kulis et al. [9] extended the LSH to the case when the similarity function is an arbitrary kernel function κ: $D(p, q) = \kappa(p, q) = \phi(p)^T \phi(q)$. Given an input feature vector x, the problem is then to design a specific LSH function over the feature vector $\phi(x)$, where $\phi(x)$ is some unknown embedding function. For this purpose, the authors proposed to construct the LSH function as: $h(\phi(x)) = sign(r^T \phi(x))$, where r is a random hyperplane drawn from $N(0, I)$ and is computed as a weighted sum of a subset of the database feature vectors. Since the newly derived $h(\phi(x))$ satisfies the LSH property (i.e., $Pr_{h \in H}[h(p) = h(q)] = D(p, q)$), the new indexing scheme is thus capable of performing similarity searching in a sub-linear time complexity, while being useful to the cases of kernelized data.

For summary, the hashing-based approaches gives a great advantage of search time efficiency, but the main drawback is the use of a huge amount of memory to construct the hash tables. In addition, as argued in [2], the search precision would be a problematic because the "good" nearest neighbors could be hashed into many adjacent buckets, making the access to single hash bin insufficient to recover the good answers. The clustering-based approaches have shown to give quite good performance in a wide range of feature types and data sizes [12], [13]. The main disadvantage is the time-consuming of the process of tree construction. The space-partition-based approaches, particularly to the KD-tree-based indexing algorithms, seem to be a proper choice for all aspects of search precision, search speedup, and tree construction time. However, for all these approaches, the performance for ENN search is still limited. In this work, we propose a linked-node m-ary tree (LM-tree) indexing algorithm, which works rather well for both the ANN and ENN tasks. Three main contributions are attributed to the proposed LM-tree. First, a new 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 presented to deal with the ANN task, in combination with the use of multiple randomized LM-trees. Our experiments carried out on one million SIFT features show that the proposed method gives a significant improvement in search performance, compared to the aforementioned indexing algorithms.

The rest of this paper is organized as follows. The proposed indexing algorithm is presented in great details in Section 2. Experimental results are presented in Section 3. We conclude the paper in Section 4.

2 The proposed algorithm

2.1 Construction of the LM-tree

Given a dataset X composing of feature vectors or points in D-dimensional space R^D, we present in the following section an indexing structure to index the dataset X supporting efficient proximity searching. For better presentation of our approach, we use the notation $\mathbf{p}$ as a point in the R^D feature space, and p_i as the i^{th} component of the point $\mathbf{p}$ $(1 \leq i \leq D)$. We also denote $p = (p_{i_1}, p_{i_2})$ as a point in $2D$ space. Before constructing the LM-tree, the dataset X is normalized by aligning it to the principal axes obtained from PCA analysis. Note that, we perform no dimension reduction in this step. In stead, PCA analysis is only used to align the data via its principle axes. Next, the LM-tree is constructed by recursively partitioning dataset X into m roughly equal-sized subsets as follows:

- Sort the axes in decreasing order of variance.

– Choose randomly two axes, i_1 and i_2, from the first L highest variance axes ($L < D$).
– Conceptually 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 a 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 equal sub-partitions:$(0, \phi_{t_1}] \cup (\phi_{t_1}, \phi_{t_2}] \cup \ldots \cup (\phi_{t_m}, 360]$.
– Partition the set X into m subsets $\{X_k\}_{k=1}^{m}$ corresponding to m angle sub-partitions obtained in the previous step.

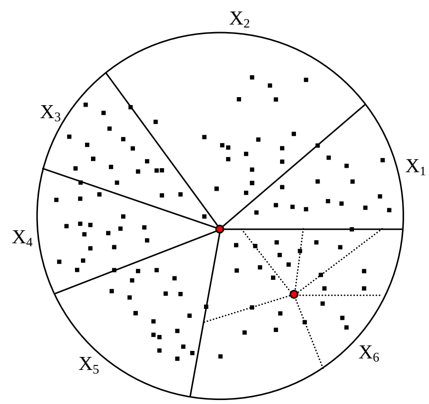


Fig. 1. Illustration of the iterative process of data partitioning in 2D space: the first partitioning applied for the entire dataset X, and the second partitioning applied for the subset X_6 (the branching factor $m = 6$).

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 about the split: split axes (i.e., i_1 and i_2), split centroid (c_{i_1}, c_{i_2}), split angles $\{\phi_{t_k}\}_{k=1}^{m}$, and 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 direct 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 associated to the child node T_k until the number of data points in each node falls below a pre-defined threshold L_{max}. In this way, only the leaf nodes contain the actual data points, and the internal nodes keep only the information about the splits. This leads to a minimal utilization of memory space with a cost of truly linear $O(n)$. It is worth pointing that each time when a partition is proceeded, two highest variance axes of the corresponding set are employed. This is contrast to many existing tree-based techniques where they often employ only one axis to partition the data. Consequently, as argued in [15], considering a high-dimensional feature space, such as 128D SIFT features, the total number of axes involved in the tree construction is rather limited, making any pruning rules inefficient and the tree less discriminative for later usage of searching. Naturally, more the number of principal axes involved in partitioning the data, more benefit we achieve for both efficiency and precision search. Figure 1 illustrates the first and second levels of the LM-tree construction with a branching factor $m = 6$.

2.2 Exact nearest neighbor search in the LM-tree

Exact nearest neighbor search in the LM-tree is proceeded using a branch-and-bound algorithm. Given a query point $\mathbf{q}$, we first project $\mathbf{q}$ into a new space using the principal axes as we have processed in the LM-tree construction. Next, starting from the root, we traverse down the tree, and use the split information stored at each node to choose the best child node for further exploration. Particularly, 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: $\phi_{qu} = \arctan(q_{i_1} - c_{i_1}, q_{i_2} - c_{i_2})$. Next, binary search is applied for the query angle ϕ_{qu} over the sequence $\{\phi_{t_k}\}_{k=1}^m$ to choose the closest child node of u from $\mathbf{q}$ for further exploration. This process continues until a leaf node is reached, and then partial distance search (PDS) [11] is applied for the points contained in the leaf. Backtracking is then invoked to explore the rest of the tree. Each time when we are positioned at some 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 the following section, we present a novelty rule to compute efficiently the lower bound. Our pruning rule is developed from that presented in the principal axis tree (PAT) [11]. PAT is a generalization of KD-tree in that the page regions are hyper-polygons rather than hyper-rectangles, and the pruning rule is recursively computed based on the law of cosines. The disadvantages of this pruning rule are expensive cost of computation (i.e., $O(D)$) and being inefficient when working on high-dimensional space due to the fact that only one axis is employed at each partition. As our algorithm 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. Particularly, we have incorporated two following major advantages for the proposed pruning rule:

- The lower bound is computed as simple as in 2D space, regardless of how large the dimensionality D is. Therefore, the computation cost is just $O(2)$ instead of $O(D)$ as in the case of PAT.
- The magnitude of the proposed lower bound is significantly greater than that of using PAT. This makes the proposed pruning rule work efficiently.

We now return to the description of computing the lower bound. Let u be the node in the LM-tree at which we are positioned, and T_k be one of the children of u which is going to be searched, and $p_k = (p_{i_1}^k, p_{i_2}^k)$ be the k^{th} split point corresponding to the child node T_k (see Figure 2). The lower bound, $LB(\mathbf{q}, T_k)$, from $\mathbf{q}$ to the child node T_k is recursively computed from $LB(\mathbf{q}, u)$ as follows:

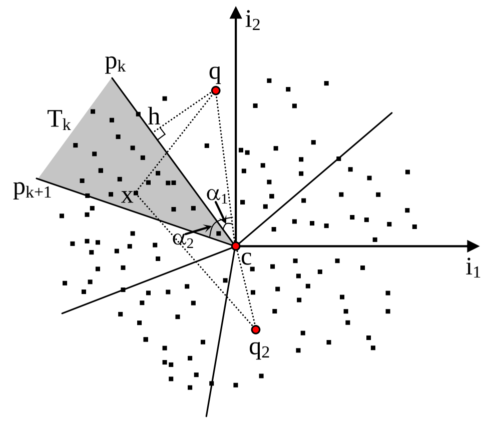


Fig. 2. Illustration of computing the lower bound in 2D space.

- 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 one of two angles α_1 and α_2 is smaller than 90^0, we have the following fact due to the rule of cosines [11]:

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

  where x is any points 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} depending of if $\alpha_1 < \alpha_2$ or not, respectively. Then, we applied the rule of lower bound computation in PAT in 2D space as follows:

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

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

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

  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, we then treat the point $\mathbf{c} = (q_1, q_2, \ldots, c_{i_1}, \ldots, c_{i_2}, \ldots, q_{D-1}, q_D)$ in place of $\mathbf{q}$ in the means of lower bound computation to the descendant of T_k.

As the lower bound $LB(\mathbf{q}, T_k)$ is recursively computed from $LB(\mathbf{q}, u)$, it is needed to set an initial value 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, therefore: $LB(\mathbf{q}, T_k) \leftarrow LB(\mathbf{q}, u)$.

2.3 Approximate nearest neighbor search in the LM-tree

In some cases when exact nearest neighbor (ENN) search is not a crucial need, approximate nearest neighbor (ANN) search is an excellently alternative choice as the precision search could be slightly degraded but we can gain a speedup of hundreds times compared to the brute-force search. In this section, we describe the use of the LM-tree to deal with the ANN task. Particularly, ANN search is proceeded by constructing multiple randomized LM-trees to account for different viewpoints of the data. The idea of using multiple randomized trees for ANN search is originally presented in [15], where the authors proposed to construct multiple randomized KD-trees. This technique has been then incorporated with the priority search and successfully used in many other tree-based structures such as hierarchical clustering trees, K-means trees, and KD-trees [12], [13]. Although the priority search is shown to give better search performance, it is certainly subjected to high cost of computation 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 avoid using the priority queue. The basic idea is to restrict the search space to the branches not very far from the considering path. In this way, we introduce a specific search procedure, so-called *bandwidth search*, which is proceeded by setting a search bandwidth to every intermediate node of the on going path. Particularly, let $P = \{u_1, u_2, \ldots, u_r\}$ be a considering path obtained by traversing on 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 is no need to be searched. The value b is called search bandwidth. Taking one example as shown in Figure 3, where X_6 is an intermediate node on the considering path P, then only X_1 and X_5 are the candidates for further inspection given a search bandwidth $b = 1$. There is a notable point that when the query $\mathbf{q}$ is too close to the centroid $\mathbf{c}$, all the sibling nodes of u_i should be inspected. In our experiments, this is happened at the node u_i if $d(q, c) \leq \epsilon D_{med}$, where q and c are the projected query point and centroid on the 2D plane associated to the split axes at u_i, and D_{med} is the median value of the distances between c and all projected data points associated to u_i, and ϵ is a tolerate parameter. In addition, in order to obtain a varied range of search precision, we would need a parameter E_{max} of maximum data points to be searched on a single LM-tree. As we are designing an efficient solution dedicated to ANN search, it would make sense to use an approximate pruning rule rather than an exact one. This at one hand gives a great benefit of computation cost, and on the other hand, it 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 closest to the query. In our case, we have used only the formula (4) as an approximate pruning rule.

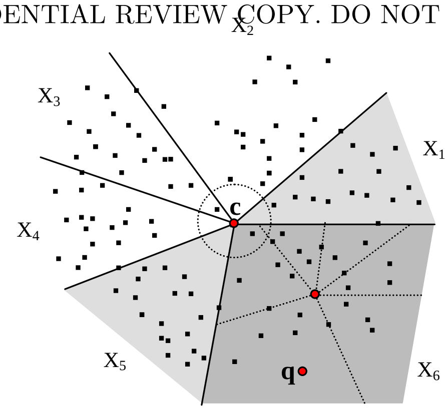


Fig. 3. Illustration of our bandwidth search with $b = 1$: X_6 is an intermediate node of the considering path, then its adjacent sibling nodes, X_1 and X_5, will be also searched; if $\mathbf{q}$ is too close to $\mathbf{c}$ (e.g., inside the circle), all the sibling nodes of u_i will be searched.

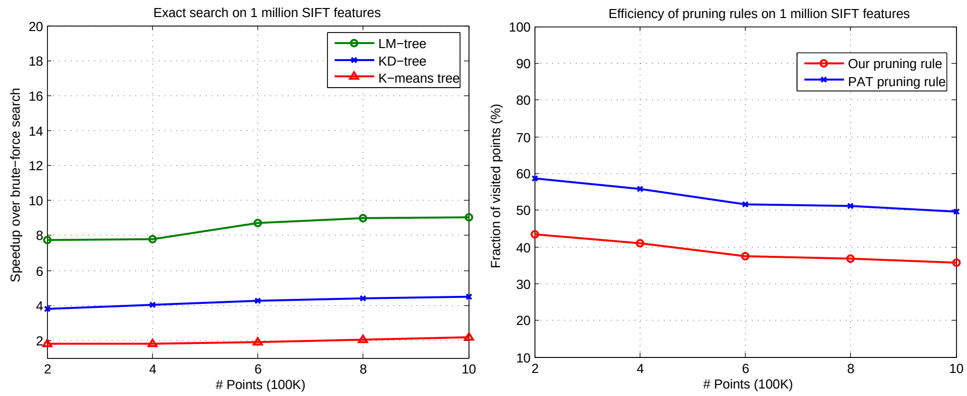
3 Experimental results

We have evaluated our system versus several representative fast proximity search systems in the literature, including randomized KD-trees (RKD-trees) [12], [15], hierarchical K-means tree (K-means tree) [12], randomized K-medoids clustering trees (RC-trees) [13], and multi-probe LSH indexing algorithm [10]. These all indexing systems have been well-implemented and widely used in the literature thanks to the open source library FLANN[1]. The source code of our system is also publicly available at this address[2]. Note that the partial distance search have been implemented in these all systems for improving the efficiency of sequence search at the leaf nodes. We have used the dataset ANN_SIFT1M from [8] for all experiments. This dataset contains a database of 1 million SIFT features, a test set of 5000 SIFT features, and a training set of 10,000 SIFT features. As no training process is required in our approach, we have therefore used the two first sets only. Following the convention of the evaluation protocol used in the literature [1], [12], [13], we have computed the precision and search time as the average measures obtained by running 1000 queries taken from the test set. To make the results independent on the machine and software configuration, the speedup factor is computed relative to the brute-force search. The details of our experiments are presented in the following sections.

3.1 ENN search evaluation

For ENN search, we have set the parameters involved in the LM-tree as follows: $L_{max} = 10$, $m = 7$, $L = 2$ (see Section 2.1). We have compared the performance of ENN search of three systems: the proposed LM-tree, the KD-tree, and the

[1] http://www.cs.ubc.ca/ mariusm/index.php/FLANN/FLANN
[2] https://sites.google.com/site/LM-tree/



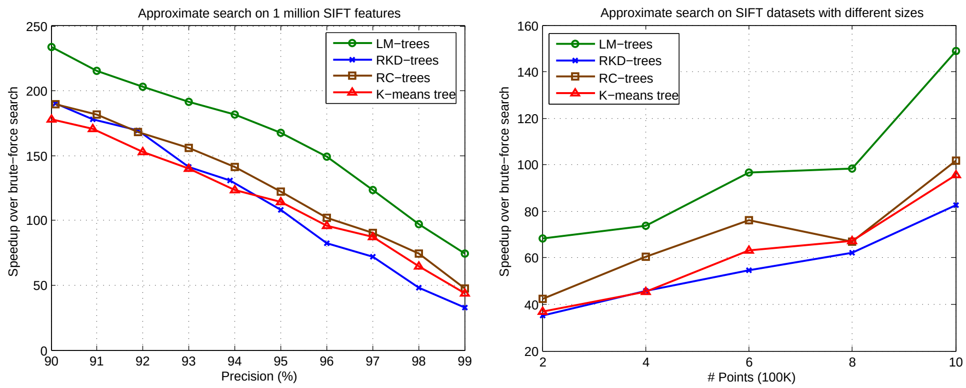
(a) (b)

Fig. 4. Exact nearest neighbor search on 1 million SIFT features: (a) Speedup over brute-force search of three systems: the proposed LM-tree, KD-tree, and K-means tree, (b) Evaluation of our pruning rule and PAT's pruning rule.

hierarchical K-means tree. Figure 4(a) shows the speedup over brute-force search of the three systems carried out on the SIFT database with different sizes. As we can notice that the LM-tree outperforms the two other systems on all tests. Taking the test with $\#Points = 1000000$, for example, the LM-tree gives a speedup of 9.1, the KD-tree gives a speedup of 4.5, and the K-means tree gives a speedup of 2.2 over the brute-force search. These results confirms the efficiency of the LM-tree for ENN search relative to the two baseline systems. For getting a more detailed analysis of the efficiency provided by our pruning rule, we present in Figure 4(b) the fraction of visited points over the size of the test of the LM-tree using our pruning rule (i.e., the rules (2) and (4) in Section 2.2), and the PAT's rule (i.e., the sole rule (2)). On average, the faction of searched points using the proposed pruning rule is almost 15% less than that of using the PAT' rule in all tests. This again support our claims about the two advantages of the proposed pruning rules compared with the original one used in PAT.

3.2 ANN search evaluation

For ANN search, we have fixed the following parameters: $L_{max} = 10$, $m = 7$, $L = 8$, $b = 1$. Four systems are participated in this evaluation, including the proposed LM-trees, RKD-trees, RC-trees, and K-means tree. We have used 8 parallel trees in the first three systems, while the last one uses a single tree because it was shown in [12] that the use of multiple K-means trees does not give better search performance. For the LM-trees, the parameters E_{max} and ϵ are empirically determined to obtain the search precision varying in $[90\%, 99\%]$. Figure 5(a) shows the search speedup versus the search precision of all the systems. As we can see that, the proposed LM-trees gives much better search performance



(a) (b)

Fig. 5. Approximate nearest neighbor search on SIFT features: (a) Speedup versus search precision of 4 systems on 1 million SIFT features; (b) Speedup of 4 systems on SIFT database with different sizes (search precision = 96%).

everywhere than the other systems and tends to perform well with respect to the increase of search precision. Taking the search precision of 95%, for example, the speedups over 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 have converted the real SIFT features to the binary vectors and tried several parameter settings (i.e., the number of hash tables, the number of multi-probe levels, and the length of the hash key) 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 instance, the speedup over brute-force search (using Hamming distance) is just 1.5. Figure 5(b) shows the search performance of all systems on the SIFT database with different sizes. In this test, the search precision is fixed at 96% for all systems. The LM-trees clearly outperforms the others and scales well to the increase of data size. The RC-trees works reasonably well except for the point $\#Points = 800K$, its search performance is noticably degraded. Three crucial factors explain for these outstanding results of the LM-trees. First, the use of the two highest variance axes for data paritioning in the LM-tree gives more discriminative representation of the data compared with 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 much of them would be eliminated after checking the lower bound. In this way, the number of data points, which will be actually searched, is retained under the pre-defined threshold E_{max}, while covering a larger number of inspected nodes, and thus increasing the chance of reaching the true nodes closest to the query. Finally, the use of bandwith search gives much of benefit in terms of computation cost compared with the priority search used in the baseline indexing systems.

4 Conclusions

In this paper, a new indexing scheme, called LM-tree, in feature vector space has been presented. The main contribution of the proposed LM-tree is three-fold. First, a new method of data decomposition is presented to construct the LM-tree. Second, a novelty elimination rule is proposed to efficiently prune the search space. Finally, a bandwidth search technique is presented to deal with the ANN task, in combination with the use of multiple randomized LM-trees. The proposed LM-tree has been validated on 1 million SIFT features, demonstrating that it works well for both ENN search and ANN search, compared with the baseline indexing algorithms. More experiments on different feature types of different domains would be performed in the future to study thoroughly the performance of the proposed LM-tree. Dynamic insertion and deletion of data points in the LM-tree would be also investigated in future works.

References

1. Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition*, CVPR'97, pages 1000–1006, 1997.
2. Jeffrey S. Beis and David G. Lowe. Indexing without invariants in 3d object recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(10):1000–1015, 1999.
3. Christian Böhm, Stefan Berchtold, and Daniel A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Comput. Surv.*, 33(3):322–373, 2001.
4. Jerome H. Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.
5. K. Fukunaga and M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *IEEE Trans. Comput.*, 24(7):750–753, 1975.
6. Volker Gaede and Oliver Günther. Multidimensional access methods. *ACM Comput. Surv.*, 30(2):170–231, June 1998.
7. Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, STOC'98, pages 604–613, 1998.
8. Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011.
9. Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing. *IEEE Trans. Pattern Anal. Mach. Intell.*, 34(6):1092–1104, 2012.
10. Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB'07, pages 950–961, 2007.
11. James McNames. A fast nearest-neighbor algorithm based on a principal axis search tree. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23(9):964–976, 2001.
12. Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

13. Marius Muja and David G. Lowe. Fast matching of binary features. In *Proceedings of the Ninth Conference on Computer and Robot Vision*, pages 404–410, 2012.
14. David Nister and Henrik Stewenius. Scalable recognition with a vocabulary tree. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR'06, pages 2161–2168, 2006.
15. Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'08)*, pages 1–8, 2008.
16. David A. White and Ramesh Jain. Similarity indexing with the ss-tree. In *Proceedings of the 12th International Conference on Data Engineering*, ICDE'96, pages 516–523, 1996.