An Efficient Indexing Scheme Based on Linked-Node m-Ary Tree Structure

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

PolytechTours, 64 Avenue Jean Portalis, 37200 Tours, France
phamtheanh@hdu.edu.vn,
{sabine.barrat,mathieu.delalandre,jean-yves.ramel}@univ-tours.fr

Abstract. Fast nearest neighbor search is a crucial need for many recognition systems. Despite the fact that a large number of indexing algorithms have been proposed in the literature, 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 to give satisfactory performance on specific benchmarks. In this work, we propose a linked-node m-ary tree (LM-tree) algorithm, which works really well for both exact and approximate nearest neighbor search. 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 introduced to explore the nodes of the LM-tree. Our experiments, applied to one million 128-dimensional SIFT features and 250000 960-dimensional GIST features, showed that the proposed algorithm gives the best search performance, compared to the aforementioned algorithms.

Keywords: Image Indexing, Locality-Sensitive Hashing, Clustering Trees.

1 Introduction

In fact, 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. A comprehensive survey of the indexing algorithms in vector space is presented by Böhm et al. [2]. These algorithms are often categorized into space-partitioning-based, clustering-based, and hashing-based approaches. We will discuss hereafter the most representative indexing algorithms.

For the space-partitioning-based approaches, KD-tree is probably argued as one of the most popular techniques [3]. The basic idea is to iteratively partition the data X into two roughly equal-sized subsets, using a hyperplane perpendicular to a split axis in a D-dimensional vector space. Two new nodes are then created corresponding to these subsets. This process is then repeated for the two subsets until the size of each subset falls below a threshold. Searching for a nearest neighbor of a given query point q is proceeded using a branch-and-bound technique whose the pruning rule works as follows: a node u is selected

A. Petrosino (Ed.): ICIAP 2013, Part I, LNCS 8156, pp. 752–762, 2013.

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 for the exact nearest neighbor (ENN) search in low-dimensional space. Several variations of the KD-tree have been investigated to deal with the approximate nearest neighbor (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 use of priority search was further improved in [12], where the authors proposed to construct multiple randomized KD-trees (RKD-trees) and principal component KD-trees (PKD-trees). The RKD-trees are constructed by selecting the split axis at random from a small set of dimensions having the highest variances. The PKD-trees are constructed in a similar manner but the data are aligned in advance to the principal axes obtained from PCA analysis. A last noticeable improvement of the KD-tree for the ENN search is principal axis tree (PAT-tree) [9]. 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 principal axis of the underlying data. Although the computation of lower bounds in the PAT-tree is quite complicated, the PAT-tree still outperforms many other indexing schemes.

For the clustering-based approaches, Fukunaga et al. [4] 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 search is then proceeded using a branch-and-bound algorithm. Muja et al. [10] extended the work of Fukunaga et al. by incorporating the priority search to deal with the ANN task. Particularly, proximity search is proceeded by traversing down the tree and always choosing the node whose cluster center 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 the increasing order of the distances to the query. This continues until a leaf node is reached followed by a sequence search for the points contained in this node. Backtracking is then invoked starting from the top node stored in the priority queue. Multiple randomized clustering trees were also explored in [11] by the same authors, where the trees are constructed by selecting the centroids at random. The experiments showed a significant improvement of search performance, compared to other indexing algorithms for many datasets.

For hashing-based approaches, Locality-Sensitive Hashing (LHS) [5] has been known as one of the most popular hashing-based methods, which can perform ANN search with a truly sub-linear time. The key idea of the LHS is to design the hash functions so that the similar points may be hashed with a high probability of collision, while the dissimilar points may likely be hashed with different keys. Given a query, 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 ANN search in a truly sub-linear time complexity. Qin Lv et al. [8] introduced *multi-probe* LSH to substantially reduce the number of hash tables, while retaining the same search precision. The basic idea is to search multiple buckets, which probably contain the potential nearest neighbors of the query. In this way, the proposed method reduces the space requirement and increases the chance of finding the true answers. Kulis et al. [7] extended the LSH to the case when the similarity function is an arbitrary kernel function κ: $D(p,q) = \kappa(p,q) = \phi(p)^T\phi(q)$, where $\phi(x)$ is some unknown embedding function. In their work, the LSH hash function is constructed 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. As the function $h(\phi(x))$ satisfies the LSH property, the new indexing scheme can perform similarity search in a sub-linear time complexity, while being useful to the cases of kernelized data.

For summary, the hashing-based approaches give a great advantage of search efficiency, but the main drawback is the use of a huge amount of memory to construct the hash tables. In addition, the search precision could be a problem because the true nearest neighbors could be hashed into many adjacent buckets, making the access to a single hash bin insufficient to recover the true answers. The clustering-based approaches have shown quite good performance in a wide range of feature types and dataset sizes [10], [11]. The main disadvantage is the expensive time of the process of tree construction. The space-partitioning-based approaches, particularly the KD-tree-based algorithms, seem to be the most appropriate solutions for all aspects of search precision, search speedup, and tree construction time.

In this work, we propose a linked-node m-ary tree (LM-tree) algorithm, which works really well for both 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. Our experiments, applied to one million 128-dimensional SIFT features and 250000 960-dimensional GIST features, showed that the proposed algorithm gives a significant improvement of search performance, compared to the baseline indexing algorithms. The rest of this paper is organized as follows: The proposed indexing algorithm is detailed 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

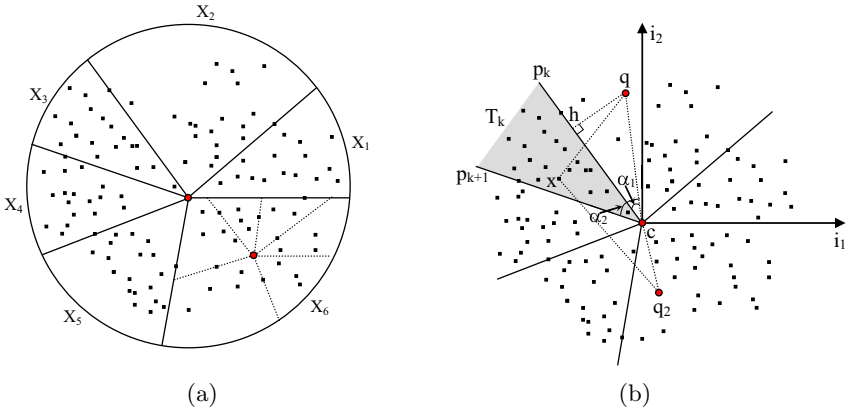
Given a dataset X composing of N feature vectors in a D-dimensional space R^D, we present, in this section, an indexing structure to index the dataset X supporting efficient proximity search. For better presentation of our approach,

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. Before constructing the LM-tree, the dataset X is normalized by aligning it to the principal axes obtained from PCA analysis. Note that no dimension reduction is performed in this step. In fact, PCA analysis is only used 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 the 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 the increasing order $(n = |X|)$, and then divide the angles into m equal-sized 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.



(a) (b)

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

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), 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 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}. It is worth pointing that

each time, when a partition is proceeded, the two highest variance axes of the corresponding dataset are employed. This is contrast to many existing tree-based indexing algorithms, where only one axis is often employed to partition the data. Consequently, as argued in [12], 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 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. Figure 1(a) 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: $\phi_{q_u} = \arctan(q_{i_1} - c_{i_1}, q_{i_2} - c_{i_2})$. Next, 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 closest to the query $\mathbf{q}$ for further exploration. This process continues until a leaf node is reached, following by partial distance search (PDS) [9] to 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 this section, we present a novel rule to compute efficiently the lower bound. Our pruning rule was inspired by the work presented in the principal axis tree (PAT) [9]. PAT is a generalization of the KD-tree, where 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 the computation cost of the complexity (i.e., $O(D)$) and being inefficient when working on a high-dimensional space, due to the fact that only one axis is employed at each partition. As 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. Particularly, we have incorporated two following major advantages for the proposed pruning rule:

- The lower bound is computed as simple as in a 2D space, regardless of how large the dimensionality D is. Therefore, the time complexity 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 in PAT. This enables the proposed pruning rule to work efficiently.

We now come back 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 $p_k = (p^k_{i_1}, p^k_{i_2})$ be the k^{th} split point corresponding to the child node T_k (see Figure 1(b)). The lower bound $LB(\mathbf{q}, T_k)$, from $\mathbf{q}$ to T_k, is recursively computed from $LB(\mathbf{q}, u)$ as follows:

- 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 [9]:

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

  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} taking into account that $\alpha_1 \leq \alpha_2$ or $\alpha_1 > \alpha_2$. Then, we applied the rule of lower bound computation in PAT in a 2D space as follows:

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

  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 angles, α_1 and α_2, are greater than 90^0 (e.g., the point q_2 in Figure 1(b)), 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 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 $LB(\mathbf{q}, u)$, an initial value has to 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.

2.3 Approximate Nearest Neighbor Search in the LM-Tree

Approximate nearest neighbor 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 was originally presented in [12], where the authors proposed to construct multiple randomized KD-trees. This technique was then incorporated with the priority search and successfully used in many other tree-based structures [10], [11]. Although the priority search was shown to give better search performance, it is certainly subjected to high computation 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 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 ongoing 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.

There is a notable point that when the projected query q is too close to the projected centroid c, all the sibling nodes of u_i should be inspected. In our case, this is happened at the 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 projected data points associated to u_i, and ϵ is a tolerate parameter. In addition, in order to obtain a varying 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 is not only reduces much of computation cost but also ensures that a larger fraction of nodes will be inspected yet 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. Particularly, we have used only the formula (4) as an approximate pruning rule as follows:

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

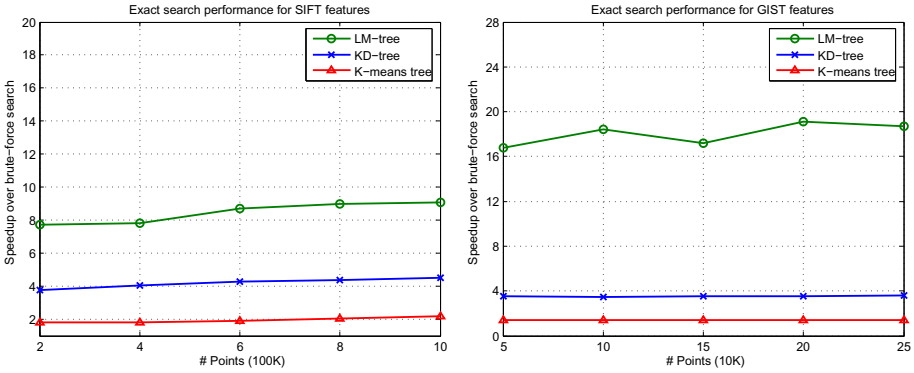
where $\kappa \geq 1$ is a pruning factor, which controls the rate of pruning the branches in the tree. This factor can be adaptively estimated during the tree construction given a specific precision and a particular dataset. However, we have fixed this value $\kappa = 2.5$ and shown, in our experiments, that it is possible to achieve satisfactory search performance on many datasets. Recall that the rule (5) requires the computation of $d(q, c)$ in a 2D space, where the point c has been already computed during the offline tree construction. The computation of this rule is thus very efficient.

3 Experimental Results

We evaluated our system versus several representative fast proximity search systems in the literature, including randomized KD-trees (RKD-trees) [10], [12], hierarchical K-means tree (K-means tree) [10], randomized K-medoids clustering trees (RC-trees) [11], and multi-probe LSH algorithm [8]. These indexing systems were well-implemented and widely used in the literature thanks to the open source FLANN library[1]. The source code of our system is also publicly

[1] `http://www.cs.ubc.ca/~mariusm/index.php/FLANN/FLANN`

available at this address[2]. Note that the partial distance search were implemented in these systems in order to improve the efficiency of sequence search at the leaf nodes. Two datasets, ANN_SIFT1M and ANN_GIST1M [6], were used for all the experiments. The ANN_SIFT1M dataset contains a database of one million 128-dimensional SIFT features and a test set of 5000 SIFT features, while the ANN_GIST1M dataset is composed of a database of one million 960-dimensional GIST features and a test set of 1000 GIST features. Since the dimensionality of the GIST feature is very high and our computer configuration is limited (i.e., Windows XP, 2.4G RAM), we were not able to load the full ANN_GIST1M dataset into memory. Consequently, we have used 250000 GIST features for search evaluation. Following the convention of the evaluation protocol used in the literature [1], [10], [11], we computed the search precision and search time as the average measures obtained by running 1000 queries taken from the test sets of the two datasets. To make the results independent on the machine and software configuration, the speedup factor is computed relative to the brute-force search.



(a) (b)

Fig. 2. ENN search performance for the SIFT (a) and GIST (b) features

3.1 ENN Search Evaluation

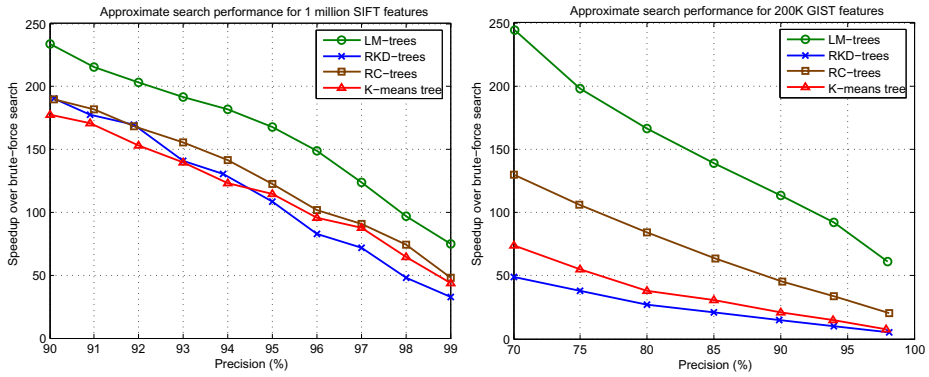
For ENN search, we set the parameters involved in the LM-tree as follows: $L_{max} = 10$, $L = 2$, and $m \in \{6, 7\}$ with respect to the GIST and SIFT datasets (see Section 2.1). We compared the performance of ENN search of three systems: the proposed LM-tree, the KD-tree, and the hierarchical K-means tree. Figure 2(a) shows the speedup over brute-force search of the three systems, applied to the SIFT datasets with different sizes. We can notice that the LM-tree outperforms the two other systems on all tests. Figure 2(b) presents the search performance of the three systems for the GIST features. The proposed LM-tree again outperforms the others and even performs far better than the SIFT features. Taking the

[2] `https://sites.google.com/site/ptalmtree/`

test where $\#Points = 150000$ on Figure 2(b), 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 ENN search relative to the two baseline systems.

3.2 ANN Search Evaluation

For ANN search, we fixed the following parameters: $L_{max} = 10$, $L = 8$, $b = 1$, and $m \in \{6, 7\}$. 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, while the last one uses a single tree because it was shown in [10] that the use of multiple K-means trees does not give better search performance. For all the systems, the parameters E_{max} and ϵ (i.e., the LM-tree) are varied to obtain a wide range of search precision. Figure 3(a) shows the search speedup versus the search precision of the four systems for 1 million SIFT features. As we can see, the proposed LM-trees algorithm gives significantly better search performance everywhere than the other systems. 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 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.



(a) (b)

Fig. 3. ANN search performance for the SIFT (a) and GIST (b) features

Figure 3(b) shows the search performance of all 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. Taking the search precision of 90%, for example, the speedups over 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. 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 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 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 bandwidth search gives much of benefit in terms of computation cost, compared to the priority search used in the baseline indexing systems.

Table 1 summarizes the main results of the proposed indexing algorithm in terms of speedup and absolute query time. The speedup is computed relative to the brute-force search as before, but the absolute query time is averaged over the 1000 queries. We report the results for both ENN and ANN search, where the search precision is set to 95% for ANN search.

Table 1. Summary of our search speedup and mean query time (second)

Results	ENN search		ANN search ($P = 95\%$)	
	SIFT 1M	GIST 250K	SIFT 1M	GIST 200K
Speedup	9.1x	18.7x	168.9x	84.7x
Mean query time	51.2	47.8	2.7	8.1

4 Conclusion and Future Works

In this paper, a new indexing algorithm 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 has been presented to construct the LM-tree. Second, a novel elimination rule has been proposed to efficiently prune the search space. Finally, a bandwidth search technique has been introduced 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 and 250000 GIST features, demonstrating that it works really well for both ENN and ANN search, compared to the baseline indexing algorithms. More experiments on different feature types 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.

Acknowledgment. This work has been supported by the Vietnam International Education Development (VIED) project.

References

1. Beis, J.S., Lowe, D.G.: Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition, CVPR 1997, pp. 1000–1006 (1997)
2. 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 (2001)
3. Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Trans. Math. Softw. 3(3), 209–226 (1977)
4. Fukunaga, K., Narendra, M.: A branch and bound algorithm for computing k-nearest neighbors. IEEE Trans. Comput. 24(7), 750–753 (1975)
5. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, STOC 1998, pp. 604–613 (1998)
6. Jégou, H., Douze, M., Schmid, C.: Product Quantization for Nearest Neighbor Search. IEEE Trans. Pattern Anal. Mach. Intell. 33(1), 117–128 (2011)
7. Kulis, B., Grauman, K.: Kernelized locality-sensitive hashing. IEEE Trans. Pattern Anal. Mach. Intell. 34(6), 1092–1104 (2012)
8. Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K.: Multi-probe lsh: efficient indexing for high-dimensional similarity search. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 950–961 (2007)
9. McNames, J.: A fast nearest-neighbor algorithm based on a principal axis search tree. IEEE Trans. Pattern Anal. Mach. Intell. 23(9), 964–976 (2001)
10. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: VISAPP International Conference on Computer Vision Theory and Applications, pp. 331–340 (2009)
11. Muja, M., Lowe, D.G.: Fast matching of binary features. In: Proceedings of the Ninth Conference on Computer and Robot Vision, pp. 404–410 (2012)
12. Silpa-Anan, C., Hartley, R.: Optimised kd-trees for fast image descriptor matching. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR 2008), pp. 1–8 (2008)