Trinary-Projection Trees for Approximate Nearest Neighbor Search

Jingdong Wang, Naiyan Wang, You Jia, Jian Li, Gang Zeng, Hongbin Zha, and Xian-Sheng Hua

Abstract—We address the problem of approximate nearest neighbor (ANN) search for visual descriptor indexing. Most spatial partition trees, such as KD trees, VP trees and so on, follow the hierarchical binary space partitioning framework. The key effort is to design different partition functions (hyperplane or hypersphere) to divide the points so that (1) the data points can be well grouped to support effective NN candidate location and (2) the partition functions can be quickly evaluated to support efficient NN candidate location. We design a trinary-projection-direction-based partition function. The trinary-projection direction is defined as a combination of a few coordinate axes with the weights being 1 or -1. We pursue the projection direction using the widely-adopted maximum variance criterion to guarantee good space partitioning and find fewer coordinate axes to guarantee efficient partition function evaluation. We present a coordinate-wise enumeration algorithm to find the principal trinary-projection direction. In addition, we provide an extension using multiple randomized trees for improved performance. We justify our approach on large scale local patch indexing and similar image search.

Index Terms—Approximate nearest neighbor search, KD trees, trinary-projection trees.

1 Introduction

Nearest neighbor (NN) search is a fundamental problem in computational geometry [12] and machine learning [42]. It also plays an important role and has various applications in computer vision and pattern recognition. The basic but essential task, content-based image and video retrieval, is a nearest neighbor problem: to find the examples that are most similar to the query in a large database. The nearest neighbor classifier, relying on NN search, is frequently employed for recognition and shape matching [18], [57]. Local feature-based object retrieval methods include the step of searching a huge database of patch descriptors for most similar descriptors [38]. Vision applications, such as 3D modeling from photo databases [44] and panorama building [8], depend on NN search for fast matching to establish the correspondences of local feature points among images. Graphics applications, such as texture synthesis [13], [27], image completion [19] and so on, also adopt NN search to quickly find the reliable image patches.

Nearest neighbor search in the d-dimensional metric space $\mathbb{R}^d$ is defined as follows: given a query $\mathbf{q}$, the goal is to find an element $\mathrm{NN}(\mathbf{q})$ from the database $\mathcal{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$ so that $\mathrm{NN}(\mathbf{q}) = \arg\min_{\mathbf{x}\in\mathcal{X}} \mathrm{dist}(\mathbf{q}, \mathbf{x})$. In this paper, we assume that $\mathbb{R}^d$ is an Euclidean space and $\mathrm{dist}(\mathbf{q}, \mathbf{x}) = \|\mathbf{q} - \mathbf{x}\|_2$, which is appropriate for most problems in computer vision. The straightforward solution, linear scan, is to compute the distance to each point whose time complexity is $O(nd)$. The time cost is too expensive for large scale high-dimensional cases. Multi-dimensional indexing methods, such as the popular KD tree [6], [17] using branch and bound or best-first techniques [3], [5], have been proposed to reduce the time of searching exact NNs. However, for high-dimensional cases it turns out that such approaches are not much more efficient (or even less efficient) than linear scan.

To overcome this issue, a lot of investigations have been made instead on approximate nearest neighbor (ANN) search. There are two basic categories of ANN search. One is error-constrained ANN search that terminates the search when the minimum distance found up to now lies in some scope around the true minimum (or desired) distance. For example, given $\varepsilon > 0$ and $\delta > 0$, $(1 + \varepsilon)$-approximate nearest neighbor search with the query $\mathbf{q}$ is to find one point $\mathbf{p}$ so that $\mathrm{dist}(\mathbf{q}, \mathbf{p}) \leqslant (1 + \varepsilon)\,\mathrm{dist}(\mathbf{q}, \mathbf{p}^*)$, with $\mathbf{p}^*$ being the true nearest neighbor, and randomized $(1+\varepsilon)$-approximate nearest neighbor search is to find such a point $\mathbf{p}$ with probability at least $1-\delta$. There are some other error-constrained ANN search problems, including randomized R-near neighbor reporting that reports each R-near neighbor of $\mathbf{q}$ with probability at least $1 - \delta$. The other one is time-constrained ANN search that terminates the search when the search reaches some prefixed time (or equivalently examines a fixed number of data points). The latter one, the focus of this paper, is more practical and gives better performance. In the following, we

- *J. Wang is with the Media Computing Group, Microsoft Research Asia, Beijing, China.*
  E-mail: jingdw@microsoft.com.
- *N. Wang is with the Department of Computer Science and Engineering, the Hong Kong University of Science and Technology, Hong Kong, China.*
  E-mail: winsty@gmail.com.
- *Y. Jia is with the Robotics Institute, Carnegie Mellon University, USA.*
  E-mail: jiayou50@gmail.com.
- *J. Li is with the Institute for Interdisciplinary Information Sciences, Tsinghua University, Beijing, China.*
  E-mail: lapordge@gmail.com.
- *G. Zeng is with the Department of Machine Intelligence, the School of Electronics Engineering and Computer Science, Peking University, Beijing, China.*
  E-mail: g.zeng@ieee.org.
- *H. Zha is with the Department of Machine Intelligence, the School of Electronics Engineering and Computer Science, Peking University, Beijing, China.*
  E-mail: zha@cis.pku.edu.cn.
- *X.-H. Hua is with Microsoft Cooperation, USA.*
  E-mail: xshua@microsoft.com.

TABLE 1
Comparison of trinary-projection (TP) trees with KD trees, PCA trees, spill trees, random projection (RP) trees, k-means trees and vantage point (VP) trees. Search order: the order of visiting data points. Branching cost: the time cost of determining which child is next accessed at the internal node. Time overhead: the extra time cost of accessing the point in a leaf node. Overall performance: the overall ANN search performance in terms of time cost and precision.

	TP tree	KD tree	PCA tree	Spill tree	RP tree	K-means tree	VP tree
search order	medium	poor	good	medium	medium	good	good
branching cost	$O(1)$	$O(1)$	$O(d)$	$O(d)$	$O(d)$	$O(d)$	$O(d)$
time overhead	low	low	high	high	high	high	high
overall performance	good	medium	medium	poor	poor	medium	poor

will review existing widely-studied ANN search algorithms, and then present the proposed approach.

1.1 Related Work

A comprehensive survey on ANN search algorithms can be found from [40]. We mainly present the review on two categories: partition trees and hashing, which are widely-used in computer vision and machine learning.

1.1.1 Partition Trees

The partition tree based approaches recursively split the space into subspaces, and organize the subspaces via a tree structure. Most approaches select hyperplanes or hyperspheres according to the distribution of data points to divide the space, and accordingly data points are partitioned into subsets. The typical partition trees include KD trees [6], [17] and its variants [3], [5], [43], box-decomposition trees (BD tree) [3], PCA tree [45], metric trees (e.g., ball trees [33], vantage point trees (VP tree) [56], random projection trees (RP tree) [10], and spill trees [29]), hierarchical k-means trees [36]. Other partition trees, such as Quadtrees [16], Octrees [55] and so on, are designed only for low-dimensional cases.

In the query stage, the branch-and-bound methodology [6] is usually adopted to search (approximate) nearest neighbors. This scheme needs to traverse the tree in the depth-first manner from the root to a leaf by evaluating the query at each internal node, and pruning some subtrees according to the evaluation and the currently-found nearest neighbors. The current state-of-the-art search strategy, priority search [3] or best-first [5], maintains a priority queue to access subtrees in order so that the data points with large probabilities being true nearest neighbors are first accessed.

Let us look at more details on KD trees, PCA trees, RP trees and spill trees, all of which use hyperplanes to split the data points. KD trees use a coordinate axis to form the partition hyperplane. In contrast, PCA trees find the principal direction using principal component analysis (PCA) to form the partition hyperplane, and spill-trees and RP trees select the best one from a set of randomly sampled projection directions. Compared with KD trees, PCA trees, RP trees and spill trees yield better space partitions and thus lead to better order for visiting the points because the partition hyperplanes are less limited and more flexible than those in KD trees. However, in the query stage, the time overhead in PCA trees, RP trees and spill trees is larger because the branching step, determining which child of an internal node is next visited, requires an inner-product operation that consists of $O(d)$ multiplications and $O(d)$ additions while it costs only $O(1)$ in KD trees. Therefore, in high-dimensional problems KD trees usually achieve better accuracy than PCA trees and spill trees within the same search time. In practice, KD trees are widely adopted for computer vision applications. A comparison of these partition trees is summarized in Table 1.

Multiple randomized KD trees, proposed in [43], generate more space partitions to improve the search performance. In the query stage, the search is performed simultaneously in the multiple trees through a shared priority queue. It is shown that the search with multiple randomized KD trees achieves significant improvement. A boosting-like algorithm is presented in [48] to learn complementary multiple trees for further performance improvement. FLANN [34], which is probably the most widely-used approach in computer vision, automatically selects one from multiple randomized KD trees and hierarchical k-means trees according to a specific database and finds the best parameters. Similarly a priority search scheme is also used in the query stage. The proposed approach in this paper can also be combined into the FLANN framework to automatically tune the parameters. This is left for future work.

1.1.2 Hashing

Locality sensitive hashing (LSH) [11], one of the typical hashing algorithms, is a method of performing ANN search in high dimensions. It could be viewed as an application of probabilistic dimension reduction of high-dimensional data. The key idea is to hash the points using several hash functions to ensure that for each function the probability of collision is much higher for points that are close to each other than those far apart. Then, one can determine near neighbors by hashing the query and retrieving elements stored in the buckets containing it. Several followup works, such as LSH forest [4] and multi-probe LSH [31], improve the search efficiency or reduce the storage cost. LSH has been widely applied to computer vision, e.g., for pose matching, contour matching, and mean shift clustering. A literature review could be found in [42]. LSH suffers from poor access order because the hash functions are achieved without exploiting the distribution of data points and the points in the same bucket (with the same hash code) are not differentiated.

Recently, a lot of research efforts have been devoted on finding good hashing functions, by using metric learning-like techniques, including optimized kernel hashing [20], learned metrics [22], learnt binary reconstruction [25], kernelized LSH [26], and shift kernel hashing [39], semi-supervised hashing [50], spectral hashing [53], and complementary hashing [54]. Such approaches get better data partitions than LSH as the hashing functions are learnt from the data, but still poorer data partitions compared with partition trees because the hierarchical way to partitioning data points in trees has better capability to group the data points than the flat way in hashing methods. In addition, it still suffers from the drawback that the points in the same bucket are not differentiated. As a result, the access order is not satisfactory and the search performance is in practice poorer than partition trees.

1.1.3 Others

There are some other methods for ANN search, such as embedding (or dimension reduction), neighborhood graph, distance based methods and so on. LSH essentially is also an embedding method, and other classes of embedding methods include Lipschitz embedding [24] and FastMap [14]. Neighborhood graph methods are another class of index structures. A neighborhood graph organizes the data with a graph structure connecting nearby data points, for example, Delaunay graph in Sa-tree [35], relative neighborhood graph [47], and k-NN (R-NN) graph [41]. The combination with KD trees shows promising performance [51]. The disadvantage of those neighborhood graph based methods lies in quite expensive computation cost for constructing the data structure. To reduce the computation cost, an algorithm of building an approximate neighborhood graph is developed in [52].

1.2 Our Approach

In this paper, we aim to improve the hyperplane-based partition trees for ANN search. The key novelty lies in designing a *trinary-projection tree*[1] to well balance search efficiency and search effectiveness, i.e., the time overhead of accessing the points and the order of accessing them. We use a combination of a few coordinate axes weighted by 1 or -1 (equivalently a combination of all the coordinate axes weighted by 1, 0 or -1), called trinary-projection direction, to form the partition hyperplane. We propose a coordinate-wise enumeration scheme based on the maximum variance criterion to efficiently pursue trinary-projection directions, guaranteeing satisfactory space partitions.

Thanks to trinary-projection directions, our approach is superior over current state-of-the-art methods. Compared with KD trees, our approach is more effective to find partition hyperplanes and hence more effective to locate NN candidates because the trinary-projection direction is capable of generating more compact data partitions. The overall time cost of evaluating the same number of leaf nodes does not increase much because the time overhead, the time cost of branching that includes projection operations, is comparable to that in KD trees. Compared with PCA trees and k-means trees, our approach is much more efficient to locate NN candidates because the projection operation in our approach only requires a sparse operation that consists of a few addition or subtraction operations while PCA trees and k-means trees conduct a more expensive projection operation that includes an inner product operation.

1. A short version appeared in our CVPR2010 paper [23].

Algorithm 1 Partition tree construction

```
Procedure PartitionTreeConstruct(list pointList)
 1. if pointList.empty() = true then
 2.     return null;
 3. else
        /* Select the partition direction */
 4.     direction ← SelectPartitionDirection(pointList);
        /* Sort pointList and choose median as the pivot element */
 5.     select median by direction from pointList;
        /* Create nodes and construct subtrees */
 6.     treeNode node;
 7.     node.partitiondirection ← direction;
 8.     node.partitionvalue ← pointList[median];
 9.     node.left ← PartitionTreeConstruct(points in pointList before median);
10.     node.right ← PartitionTreeConstruct(points in pointList not before median);
11.     return node;
12. end if
```

2 Definition

In this section, we give a brief introduction to partition trees and partition functions, and define the trinary-projection direction that combines the coordinate axes using trinary weights.

2.1 Partition Tree

A partition tree is a tree structure that is formed by recursively splitting the space and aims to organize the data points in a hierarchical manner. Each node of the tree is associated with a region in the space, called a cell. These cells define a hierarchical decomposition of the space. The root node r is associated with the whole set of data points $\mathcal{X}$. Each internal node v is associated with a subset of data points $\mathcal{X}_v$ that lie in the cell of the node. It has two child nodes $\text{left}(v)$ and $\text{right}(v)$, which correspond to two disjoint subsets of data points $\mathcal{X}_{\text{left}(v)}$ and $\mathcal{X}_{\text{right}(v)}$. The leaf node l may be associated with a subset of data points or only contain a single point. The pseudo-code of constructing a partition tree (with hyperplanes for space division) is presented in Algorithm 1.

The key problem in constructing partition trees is to find a partition function for each internal node. For approximate nearest neighbor search, the partition function determines if the space is well decomposed and accordingly affects the order of accessing the points. On the other hand, the time complexity of evaluating partition functions determines the search efficiency because traversing the tree involves executing a lot of branching operations in internal nodes for which we need to evaluate the partition functions.

2.2 Linear Partition Function

The partition function can generally be written as $f(\mathbf{x}; \boldsymbol{\theta})$ with $\boldsymbol{\theta}$ being function parameters. Depending on the function design, partition trees can be categorized into binary partition trees, including KD trees, PCA trees, RP trees, VP trees and so on, and multi-way partition trees, including hierarchical k-means trees, quadtrees and so on. This paper mainly focuses on binary partition trees. The partition function for KD trees, PCA trees, and RP trees, is essentially a linear function, $f(\mathbf{x};\boldsymbol{\theta}) = f(\mathbf{x};\mathbf{w},b) = \mathbf{w}^T\mathbf{x} - b$, where $\mathbf{w}$ is the projection direction (also called partition direction) and b is the partition value. The space is partitioned by the hyperplane $f(\mathbf{x};\mathbf{w},b) = 0$. To determine which of the two sides a particular point lies on, we simply evaluate the sign of the partition function value at the point. The evaluation of such a partition function generally requires $O(d)$ multiplication operations and $O(d)$ addition operations. Particularly, its evaluation in KD trees is much cheaper and costs only $O(1)$, independent of the dimension d because only one entry in $\mathbf{w}$ in KD trees is 1, and all the other entries are 0. In VP trees, $f(\mathbf{x};\boldsymbol{\theta}) = f(\mathbf{x};\mathbf{c},r) = \|\mathbf{c}-\mathbf{x}\|_2 - r$. In this paper, we study the linear partition function and aim to find one function that is able to generate compact space partitions and can be efficiently evaluated.

2.3 Trinary Projection

The main idea of trinary projection is to make use of a linear combination of coordinate axes with trinary-valued weights to determine the linear partition function $f(\mathbf{x};\boldsymbol{\theta}) = f(\mathbf{x};\mathbf{w},b) = \mathbf{w}^T\mathbf{x} - b$. Here, $\mathbf{w} = [w_1 \cdots w_i \cdots w_d]^T$ with w_i being 1, 0, or -1 is called the *trinary-projection direction*. One of the advantages is that it takes $O(\|\mathbf{w}\|_0)$ addition (subtraction) operations to evaluate $f(\mathbf{x};\boldsymbol{\theta})$, which is computationally cheap. The value b can be chosen as the mean or the median of the projection values of the points along the projection direction $\mathbf{w}$.

Moreover, trinary projection is able to produce more compact space partitions compared with KD trees using coordinate axes to directly form the partition hyperplane because the partition function formed from the trinary-projection direction is more flexible. The projection direction in KD trees can be regarded as a special trinary-projection direction, only selecting one coordinate axis, equivalently, $\|\mathbf{w}\|_0 = 1$. An illustration of partitioning with a KD tree and a TP tree is shown in Figure 1.

2.4 Principal Trinary-Projection Tree

A *principal trinary-projection tree* is a partition tree, in which the direction $\mathbf{w}$ used in the partition function is the principal trinary projection direction that leads to compact space partitions. The principal trinary-projection direction $\mathbf{p}$ is a trinary-projection direction along which the variance of the normalized projections of the data points is maximized. The mathematical formulation is as follows,

$$\mathbf{p} = \arg\max_{\mathbf{w}\in\mathcal{T}} h(\mathbf{w}) \quad (1)$$
$$= \arg\max_{\mathbf{w}\in\mathcal{T}} \mathrm{Var}_{\mathbf{x}\in\tilde{\mathcal{X}}}[\|\mathbf{w}\|_2^{-1}\mathbf{w}^T\mathbf{x}], \quad (2)$$

where $\mathcal{T}$ is the whole set of trinary projection directions, $\tilde{\mathcal{X}}$ is the set of the points that is to be split, and

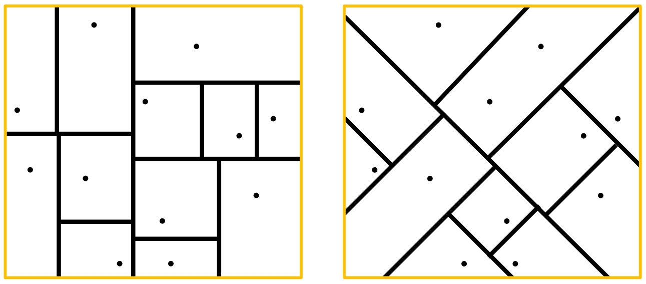


Fig. 1. Illustrating a KD tree and a TP tree in the 2D space. Using a KD tree, coordinate axes are directly used to formulate the projection directions to partition the space, as shown in (a), while using a TP tree the space can be partitioned more flexibly and can be like the partitions shown in both (a) and (b).

$\mathrm{Var}_{\mathbf{x}\in\tilde{\mathcal{X}}}[\|\mathbf{w}\|_2^{-1}\mathbf{w}^T\mathbf{x}]$ is the variance of the projections along the normalized direction $\|\mathbf{w}\|_2^{-1}\mathbf{w}$.

Partitioning along the projection direction with a large variance is known to be a competent method to partition the data [6], [17], [45]. Consider two projection directions $\mathbf{p}_1$ and $\mathbf{p}_2$ with variances c_1 and c_2, where $c_1 > c_2$ and c_1 is the largest variance. The larger variance over the data points corresponding to the two partitions resulting from $\mathbf{p}_2$ is likely to be close to c_1, while that resulting from $\mathbf{p}_1$ is likely to be much smaller than c_1. As a result, the two partitions obtained from the projection direction with a larger variance tend to be more compact, and thus, roughly speaking, the distances between the points within one partition are smaller on average. On the other hand, as pointed out in [45], the ball centered at the query point with the radius being the distance to the current best nearest neighbor intersects the partition hyperplane formed by the projection direction with a larger variance less often, and hence fewer nodes are visited in traversing the tree on average. Based on the above rationale, we adopt the maximum variance criterion to determine the partition function. We would like to remark that other criteria may work well in some cases.

3 Construction

The procedure of constructing a principal trinary projection tree is described as follows. It starts from the root that is associated with the whole set of points, and divides the points into two disjoint subsets using a partition function, each corresponding to a child node. The process is recursively performed on each new node and finally forms a tree, in which each leaf node may contain a certain number of points. This procedure is almost the same to that for constructing a KD tree, and the only difference lies in the partition function construction.

Finding the optimal principal trinary-projection direction, i.e., solving Equation 2, is a combinatorial optimization problem, which might be NP-hard. The solution space $\mathcal{T}$ consists of $\frac{3^d-1}{2}$ elements, which grows exponentially with respect to the dimension d. We can simply turn the problem into the problem of maximizing the joint probability over a Markov random field, and then optimize it using the iterated conditional modes approach [7] or other approaches, such as belief propagation. However, the weights are highly coupled together, making those solutions unsuitable.

In the following, we first present a coordinate-wise enumeration algorithm to find an approximate principal trinary-projection direction. Then we give a brief review of the cardinality-wise enumeration algorithm proposed in [23]. Last, we propose an extension to multiple randomized principal trinary projection trees.

3.1 Coordinate-Wise Enumeration

One can decompose the problem formulated in Equation 2 into a sequence of d subproblems, $\{P_1, \cdots, P_i, \cdots, P_d\}$. The subproblem P_i is defined as follows,

$$\max\ h(\mathbf{w}) \tag{3}$$
$$\text{s.t. } w_j = 0, \forall j \in \{i+1, \cdots, d\} \tag{4}$$
$$\mathbf{w} \in \mathcal{T}. \tag{5}$$

It can be easily validated that the subproblem P_d is equivalent to the problem in Equation 2. Intuitively, the subproblem P_i only exploits the first i coordinate axes to form a trinary-projection direction. We denote the feasible solutions of P_i by a set $\mathcal{T}_i$, where $\mathcal{T}_i = \{\mathbf{w} | \|\mathbf{E}_i \mathbf{w}\|_1 = 0, \mathbf{w} \in \mathcal{T}\}$ and $\mathbf{E}_i$ is a diagonal matrix with the first i diagonal entries being 0 and the remaining diagonal entries being 1.

It is apparent that $\mathcal{T}_{i-1} \subset \mathcal{T}_i$. Let $\mathcal{U}_i = \mathcal{T}_i - \mathcal{T}_{i-1}$. From the definition, $\mathcal{U}_i$ can be generated from $\mathcal{T}_{i-1}$, $\mathcal{U}_i = \{\mathbf{w} | w_i \in \{1, -1\}, \mathbf{w} \odot \mathbf{e}_i \in \mathcal{T}_{i-1}\}$, where $\odot$ is a coordinate-wise product operator and $\|\mathbf{e}_i\|_1 = d-1$ and its i-th entry is equal to 0. Intuitively, for each solution $\mathbf{w} \in \mathcal{T}_{i-1}$, we can set w_i to be 1 and -1 to form $\mathcal{U}_i$. This suggests that we incrementally enumerate the feasible solutions $\mathcal{T}$ in order of increasing i from $\mathcal{T}_1$ to $\mathcal{T}_d$, which is called coordinate-wise enumeration.

Searching for the true principal trinary-projection direction with this incremental enumeration manner still requires to check all the $\frac{3^d-1}{2}$ feasible solutions. The expensive time cost makes it impractical to find the optimal solution. Instead, we are willing to settle for an approximate principal trinary-projection direction, and enumerate a subset of possibly better feasible solutions coordinate-wisely. We show that there is an optimal order of exploiting coordinate axes to form the feasible solutions. Moreover, we analyze the performance of our approximation algorithm by providing proper upper bounds of the difference between the approximate solution and the optimal solution.

Consider the d subproblems $\{P_1, \cdots, P_d\}$. It can easily be validated that $\max_{\mathbf{w} \in \mathcal{T}_1} h(\mathbf{w}) \leqslant \max_{\mathbf{w} \in \mathcal{T}_2} h(\mathbf{w}) \leqslant \cdots \leqslant \max_{\mathbf{w} \in \mathcal{T}_d} h(\mathbf{w})$, since $\mathcal{T}_1 \subset \mathcal{T}_2 \subset \cdots \subset \mathcal{T}_d$. In addition, $\max_{\mathbf{w} \in \mathcal{T}_i} h(\mathbf{w})$ is lower bounded, which is stated in the following theorem. The proof of the theorem can be found from the supplemental material.

Theorem 1. *For coordinate-wise enumeration with the order of coordinate axes $\mathcal{B} = \{\mathbf{b}_1, \ldots, \mathbf{b}_d\}$, the optimal result of the problem P_i is lower bounded:* $\max_{\mathbf{w} \in \mathcal{T}_i} h(\mathbf{w}) \geqslant \max_{\mathbf{w} \in \mathcal{T}_d} h(\mathbf{w}) - \sum_{j=i+1}^d h(\mathbf{b}_j)$.

Algorithm 2 Coordinate-wise enumeration

```
    /* $D$: the dimension; topD: the number of used coordinate axes; $r$: the threshold of the ratio between the variance of the best direction and the maximum variance gain; topK: the number of directions kept in each enumeration; */
Procedure CoordinateWiseEnumeration(list pointList)
    /* Compute variances for all coordinate axes */
 1. axesVariances[$1 \cdots D$] $\leftarrow$ ComputeVariances(pointList);
    /* Record the possible maximum variance gain */
 2. remainedVariance $\leftarrow$ ComputeSum(axesVariances[$1 \cdots D$]);
    /* Sort the axes in the variance-increasing order */
 3. C[$1 \cdots D$] $\leftarrow$ SortAxes(axesVariances[$1 \cdots D$]);
 4. directions $\leftarrow \emptyset$;
 5. bestDirection $\leftarrow$ null;
 6. $i \leftarrow 0$;
 7. while ($i <$ topD and directionVariances(bestDirection) / remainedVariance $< r$) do
 8.     $i \leftarrow i + 1$;
 9.     directions $\leftarrow$ MergeEnumerations(directions, directions + C[$i$], directions - C[$i$]);
10.     directionVariances $\leftarrow$ ComputeVariances(directions, axesVariances [$1 \cdots D$]);
        /* Keep topK directions */
11.     directions $\leftarrow$ Prune(directions, topK);
12.     bestDirection $\leftarrow$ FindMaximum(directions, directionVariances);
        /* Update the possible maximum variance gain */
13.     remainedVariance $\leftarrow$ remainedVariance - axesVariances[C[$i$]];
14. end while
15. return bestDirection;
```

Let us consider ordered coordinate axes $\mathcal{B}^* = \{\mathbf{b}_1^*, \ldots, \mathbf{b}_d^*\}$, a permutation of $\{\mathbf{b}_1, \ldots, \mathbf{b}_d\}$, where $h(\mathbf{b}_1^*) \geqslant \cdots \geqslant h(\mathbf{b}_d^*)$, and another permutation $\mathcal{B}' = \{\mathbf{b}'_1, \ldots, \mathbf{b}'_d\}$. We can easily prove that $\sum_{j=i+1}^d h(\mathbf{b}_j^*) \leqslant \sum_{j=i+1}^d h(\mathbf{b}'_j)$ for any $i \in \{1, \cdots, d-1\}$. This implies that the lower bound of $\max_{\mathbf{w} \in \mathcal{T}_i^*} h(\mathbf{w})$ for $\mathcal{B}^*$ is not less than that of $\max_{\mathbf{w} \in \mathcal{T}'_i} h(\mathbf{w})$ for $\mathcal{B}'$, i.e., $\max_{\mathbf{w} \in \mathcal{T}_d} h(\mathbf{w}) - \sum_{j=i+1}^d h(\mathbf{b}_j^*) \geqslant \max_{\mathbf{w} \in \mathcal{T}_d} h(\mathbf{w}) - \sum_{j=i+1}^d h(\mathbf{b}'_j)$. In other words, the optimal result of P_i with respect to $\mathcal{B}^*$ has the largest lower bound, and hence is potentially a better solution. This suggests an optimal coordinate-wise enumeration scheme exploiting the coordinate axes in order of decreasing variances, which is able to find better solutions quickly.

In the optimal coordinate-wise enumeration scheme, we have a property how the solution of P_i approximates the global principal trinary-projection direction (the one of P_d) as stated in Corollary 1, which is derived from Theorem 1.

Corollary 1. *The variance deficiency, i.e., the difference of the variance along the principal trinary projection direction for the subproblem P_i from that along the global principal trinary-projection direction, is upper bounded:* $\max_{\mathbf{w} \in \mathcal{T}} h(\mathbf{w}) - \max_{\mathbf{w} \in \mathcal{T}_i^*} h(\mathbf{w}) \leqslant \sum_{j=i+1}^d h(\mathbf{b}_j^*)$.

This corollary indicates that we can avoid an expensive process of computing the global solution and only conduct a cheap process ($O(d)$) of computing the summation of the variances over all the coordinate axes to estimate the approximation degree. This suggests that we can early terminate the enumeration at the problem P_i if the deficiency upper bound $\sum_{j=i+1}^d h(\mathbf{b}_j^*)$ is small enough.

The early termination scheme reduces the time cost from

$O(3^d)$ to $O(3^{\bar{d}})$ if the enumeration process stops at the problem $P_{\bar{d}}$ for which the variance deficiency is small enough or $\bar{d}$ reaches a fixed number. But the time cost is still expensive. To reduce the cost furthermore, we introduce a practical coordinate-wise enumeration scheme. Let us compare the number of the feasible solutions, $\mathcal{T}_{i-1}$ and $\mathcal{T}_i$, for subproblems P_{i-1} and P_i, respectively. We have that $|\mathcal{T}_i| = |\mathcal{T}_{i-1} \cup \mathcal{U}_i| = |\mathcal{T}_{i-1}| + |\mathcal{U}_i|$. This suggests a speedup algorithm by generating a smaller set $\mathcal{U}_i$. Recall that $\mathcal{U}_i = \{\mathbf{w} \mid w_i \in \{1,-1\}, \mathbf{w} \odot \mathbf{e}_i \in \mathcal{T}_{i-1}\}$ and $|\mathcal{U}_i| = 2|\mathcal{T}_{i-1}|$. We use a subset of $\mathcal{T}_{i-1}$ to form $\mathcal{U}_i$. Specifically, we select a small number (g) of leading trinary-projection directions with the largest variances, $\widetilde{\mathcal{T}}_{i-1}$, to form $\tilde{\mathcal{U}}_i = \{\mathbf{w} | w_i \in \{1,-1\}, \mathbf{w} \odot \mathbf{e}_i \in \tilde{\mathcal{T}}_{i-1}\}$. In this way, the time complexity is reduced from an exponential one $O(3^{\bar{d}})$ to a polynomial one $O(g\bar{d})$. The pseudo-code of coordinate-wise enumeration is presented in Algorithm 2.

3.2 Cardinality-Wise Enumeration

We give a brief review of the approach presented in [23]. One can decompose the problem of maximizing $h(\mathbf{w})$ into a sequence of subproblems, $\{P_1, \cdots, P_d\}$, where P_i corresponds to maximizing $h(\mathbf{w})$ subject to $\|\mathbf{w}\|_1 \leqslant i$ and $\mathbf{w} \in \mathcal{T}$. It can be easily validated that the subproblem P_d is equivalent to the problem in Equation 2. We denote the feasible solutions of the subproblem P_i by $\mathcal{T}_i = \{\mathbf{w} \mid \|\mathbf{w}\|_1 \leqslant i, \mathbf{w} \in \mathcal{T}\}$. It can be easily shown that $\mathcal{T}_i = \mathcal{T}_{i-1} \cup \mathcal{U}_i$ with $\mathcal{U}_i \triangleq \{\mathbf{w} \mid \|\mathbf{w}\|_1 = i\}$. $\mathcal{U}_i$ can be easily generated from $\mathcal{U}_{i-1}$, $\mathcal{U}_i = \{\mathbf{w} \mid \|\mathbf{w} - \bar{\mathbf{w}}\| = 1, \exists \bar{\mathbf{w}} \in \mathcal{U}_{i-1}\}$. This suggests the so-called cardinality-wise enumeration scheme, enumerating the feasible solutions in order of increasing the number of the coordinate axes that are used to form the partition function.

Instead of solving the expensive problem P_d, the approach in [23] solves the subproblem $P_{\bar{d}}$ to get an approximate solution. Here we present Theorem 2 to show the approximation quality of the approximate solution $P_{\bar{d}}$, compared with the optimal solution. The proof of the theorem can be found from the supplemental material.

Theorem 2. *For cardinality-wise enumeration, the optimal result of the problem $P_{\bar{d}}$ is lower bounded:* $\max_{\mathbf{w} \in \mathcal{T}_{\bar{d}}} h(\mathbf{w}) \geqslant \max_{\mathbf{w} \in \mathcal{T}_d} h(\mathbf{w}) - \sum_{i=\bar{d}+1}^{d} h(\mathbf{b}_i^*)$, *where* $\{\mathbf{b}_1^*, \ldots, \mathbf{b}_d^*\}$ *is a permutation of coordinate axes* $\{\mathbf{b}_1, \ldots, \mathbf{b}_d\}$ *and* $h(\mathbf{b}_1^*) \geqslant \cdots \geqslant h(\mathbf{b}_d^*)$.

The approximate solution reduces the time cost from an exponential one $O(3^d)$ to a polynomial one $O(d^{\bar{d}})$, which is still very large. A further approximation method is introduced in [23]. In that method, we select a small number ($\bar{d}$) of leading coordinate axes with the largest variances and keep only g trinary-projection direction candidates, to form $\mathcal{U}_i$, which results in small time cost $O(g\bar{d}^2)$.

3.3 Multiple Randomized TP Trees

We propose to make use of multiple randomized TP trees to organize the points. It has been shown that the simultaneous search over multiple trees through a shared priority queue is superior to the priority search over a single tree [23], [43].

The simple approach for constructing a randomized TP tree is to randomly sample the weights $\mathbf{w}$. For example, one can sample the weight w_i for each coordinate axis from $\{-1, 0, 1\}$ with probabilities $\{p_{-1}, p_0, p_1\}$. The probabilities could be the same, $p_{-1} = p_0 = p_1 = \frac{1}{3}$. Or they can be computed from a Gaussian-like distribution $p_a \propto \frac{1}{\sigma\sqrt{2\pi}} \exp(-\frac{a^2}{2\sigma^2})$, where $\alpha = -1, 0, 1$ and σ is used to adjust the weights, further normalized so that $p_{-1} + p_0 + p_1 = 1$. The latter one can be regarded as an approximate to the random projection tree [10]. One drawback is that those approaches are independent to the data distribution. Instead, we propose to modify the coordinate-wise enumeration scheme to generate randomized multiple TP trees.

One candidate scheme is to randomly sample one from several top candidate principal trinary-projection directions collected from the coordinate-wise enumeration scheme. This straightforward scheme is however time-consuming for generating multiple trees. Instead, we propose a coordinate-wise random enumeration scheme to sample a principal trinary-projection direction, which may result in lower search quality for a single tree, but can still guarantee high search quality due to the complementarity of multiple trees.

We permute the coordinate axes in order of decreasing variances, $\mathbf{b}_1^*, \cdots, \mathbf{b}_d^*$, where $h(\mathbf{b}_1^*) \geqslant \cdots \geqslant h(\mathbf{b}_d^*)$. The coordinate-wise random enumeration scheme is described as follows. We first randomly sample a coordinate axis from the $\bar{d}$ leading coordinate axes, forming a trinary projection direction $\mathbf{v}_1$ for the first iteration. For discussion convenience, we assume $\mathbf{v}_1 = \mathbf{b}_j^*$. Then, we sequentially consider the remaining coordinate axes $\{\mathbf{b}_1^*, \cdots, \mathbf{b}_{j-1}^*, \mathbf{b}_{j+1}^*, \cdots, \mathbf{b}_d^*\}$. We denote the trinary-projection direction by $\mathbf{v}_t$ at the t-th iteration. For the $(t+1)$-th iteration considering the next coordinate axis $\mathbf{b}$, there are three candidates for $\mathbf{v}_{t+1}$, $\mathcal{C}_t = \{\mathbf{v}_t, \mathbf{v}_t + \mathbf{b}, \mathbf{v}_t - \mathbf{b}\}$. The sampling weight for each candidate $\mathbf{c} \in \mathcal{C}_t$ is computed from the variances, $p(\mathbf{c}) = \frac{h(\mathbf{c})}{\sum_{\mathbf{w} \in \mathcal{C}_t} h(\mathbf{w})}$. In our implementation, we found that the coordinate axes with small variances contribute little to space partitioning and add a little more time cost in the query stage. So we stop the iteration when the top $\bar{d}$ coordinate axes have been considered.

4 Analysis

4.1 Construction Time Complexity

The computation required to find the partition function for a node v associated with n_v points is analyzed as follows. It consists of the computation of the covariances and the enumeration of feasible trinary projection directions. Computing the variances for all the d coordinate axes requires $O(dn_v)$ time, and computing the covariances for the top $\bar{d}$ coordinate axes requires $O(\bar{d}^2 n_v)$ time. The enumeration cost is independent of the number of points and only depends on the number of top coordinate axes $\bar{d}$, denoted by $O(e_{\bar{d}})$, which will be analyzed in detail later. The above analysis indicates that the time cost for the node v includes two parts: $O(e_{\bar{d}})$ and $O((d + \bar{d}^2)n_v)$. Consider the first part, there are $O(n)$ nodes in the tree, then the time cost contributed by the first part is $O(ne_{\bar{d}})$. Considering the tree built in the case of using the median as the partition value is a balanced tree with the height $\log n$ and the number of points in each level is n, the time cost for each level contributed by the second part is $O((d+\bar{d}^2)n)$. In summary, the total time cost is $O(n((d+\bar{d}^2)\log n + e_{\bar{d}}))$.

Let us look at the enumeration cost $e_{\bar{d}}$. Coordinate-wise enumeration totally generates $O(g\bar{d})$ candidate directions, where g is the number of the candidate directions generated for each enumeration iteration. Evaluating the variance of each one using an incremental manner takes $O(\bar{d})$ time. Thus, coordinate-wise enumeration takes $e_{\bar{d}} = O(g\bar{d}^2)$ and coordinate-wise random enumeration takes $O(\bar{d}^2)$.

4.2 Storage Cost

The tree structure needs to store the data points and partition functions for internal nodes. Generally, a partition function needs $O(d)$, e.g., for PCA-trees and random projection trees. The trinary-projection direction in our approach is sparse, and only costs $O(\bar{d})$. The total storage cost is $O(nd + mn\bar{d})$ for m trees, where nd is the cost for storing the features of data points.

4.3 Search Time Complexity

To find approximate nearest neighbors of a query point, a top-down searching procedure is performed from the root to the leaf nodes. At each internal node, it is required to inspect which side of the partition hyperplane the query point lies in, then the associated child node is accordingly accessed. The descent down process proceeds till reaching a leaf node. The data point associated with the first leaf node is the first candidate for the nearest neighbor, which is not necessarily the true nearest neighbor. It must be followed by a process of iterative search, in which more leaf nodes are searched for better candidates. The widely used scheme with high chances to find true nearest neighbors early is *priority search* so that the cells are searched in order of increasing their distances to the query point. The ANN search terminates when a fixed number of leaf nodes are accessed. The pseudo-code of the search procedure is presented in Algorithm 3.

In the following, we show the time cost for ANN search by bounding the number of accessed leaf nodes. Accessing a leaf node in the priority search requires the descent from an internal node to this leaf node, then the descent needs to check $O(\log n)$ internal nodes. Handling each internal node consists of the evaluation of the corresponding partition function, computing the lower bound of the distance to the cell that is to be inserted to the priority queue, and the insertion and extraction operations on the priority queue. The evaluation of the partition function costs only $O(\bar{d})$. Using the binomial heap for the priority queue, it takes amortized $O(1)$ time to insert and extract a cell. In our experiments, we implemented the priority queue as a binary heap. Theoretically, the insertion operation may take $O(\log n)$ time with a binary heap, but we observed that they took only $O(1)$ time on average. The computation of the lower bound of the distance of a query to a cell costs only $O(1)$ time. Therefore, assuming that one leaf node contains only one data point, the time cost when accessing N leaf nodes is $O(N\bar{d}\log n + Nd)$, where Nd is the cost of computing the distances between the query and the data points.

Algorithm 3 Partition tree query

```
Procedure PartitionTreeQuery(Point q, treeNode root)
1.  PriorityQueue queue;
2.  topElement.node ← root;
3.  topElement.distance ← 0;
4.  minDistance ← INF;
5.  accessedPointNumber = 0;
    /* maxAccessedPointNumber: the maximum number of accessed points */
6.  while accessedPointNumber < maxAccessedPointNumber do
        /* Descend down to a leaf node */
7.      while topElement.node.IsLeaf() = false do
8.          left ← topElement.node.left;
9.          right ← topElement.node.right;
10.         direction ← topElement.node.partitiondirection;
11.         value ← topElement.node.partitionvalue;
12.         projection ← q along direction;
13.         if (projection < value) then
14.             topElement.node ← left;
15.             newElement.node ← right;
16.         else
17.             topElement.node ← right;
18.             newElement.node ← left;
19.         end if
            /* Estimate the lower bound of the distance of the query to the cell */
20.         newElement.distance ← topElement.distance + (projection - value)$^2$/$\|$direction$\|_2^2$;
21.         queue.insert(newElement);
22.     end while
23.     accessedPointNumber ← accessedPointNumber + 1;
24.     currentDistance ← ComputeDistance(topElement.node.point, q);
25.     if currentDistance < minDistance then
26.         minDistance ← currentDistance;
27.         nearestNeighbor ← topElement.node.point;
28.     end if
29.     topElement ← queue.top();
30.     queue.pop();
31. end while
32. return nearestNeighbor;
```

5 Discussion

5.1 Embedding vs. Space Partition

In metric embedding, if we project the data points to randomly sampled trinary-projection directions (this is also called database-friendly random projection [1]), the distortion of the embedding is very close to 0 (all pairwise distances are approximately preserved). This is guaranteed by the theorem in [1], which is an analogue of the Johnson−Lindenstrauss lemma for random projections: *Given a set* $\mathcal{X}$ *of* n *points in* $\mathbb{R}^d$*, and* $\varepsilon, \beta > 0$*, let* $k_0 = \frac{4+2\beta}{\frac{\varepsilon^2}{2} - \frac{\varepsilon^3}{3}} \log n$ *and* $k \geqslant k_0$*. Let* $f(\mathbf{x}) = \frac{\sqrt{3}}{\sqrt{k}} \mathbf{R}^T \mathbf{x}$*, where* $\mathbf{R}$ *is a matrix of size* $d \times k$*. The entry* r_{ij} *is sampled from* $\{1, 0, -1\}$ *with the corresponding probabilities* $\{\frac{1}{6}, \frac{2}{3}, \frac{1}{6}\}$*. Then, with probability at least* $1 - n^{-\beta}$, $(1-\varepsilon)\|\mathbf{x}_1 - \mathbf{x}_2\|_2^2 \leqslant \|f(\mathbf{x}_1) - f(\mathbf{x}_2)\|_2^2 \leqslant (1+\varepsilon)\|\mathbf{x}_1 - \mathbf{x}_2\|_2^2$ *for all* $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{X}$.

In this paper, we use the trinary-projection directions for partitioning the data points rather than for metric embedding. Assume that the tree is formed by adopting the same trinary projection for the nodes in the same depth. As the goal, the points lying in the same subspace (cell) generated by the tree are expected to be compact in the embedding space. According to the above theorem that indicates that the distances computed over the embedding space can approximate the distances computed over the original space, it can be expected that those points are likely to be compact in the original space. This implies that the points in the same subspace are very likely to be near neighbors and in other words a query point lying in a subspace can potentially find the nearest neighbors from the subspace. To obtain a better distance approximation, the projection direction of each internal node can be estimated adaptively from the points associated with the node, discarding the constraint of using the same trinary-projection direction for the nodes with the same depth.

5.2 Partition Value

Let us discuss the choice of the partition value b in the partition function $f(\mathbf{x}; \mathbf{w}, b)$. The partition value is usually determined according to the projection values. Several choices, such as mean, median and bisector, have been discussed in [45]. It is shown that selecting the median value as the partition value, resulting in a balanced tree [17]. However, it should be noted that there is no guarantee that one choice will always generate the optimal search performance in all cases. A learning-based approach is proposed in [9] to determine the partition value. In our experiments, we find that adopting the mean as the partition value (the resulting TP tree is nearly balanced in our experiments) produces similar (better in some cases) search results and takes less construction time.

5.3 Orthogonal TP Tree

The priority search procedure relies on maintaining a priority queue so that we can access the cells in order efficiently. The lower bound of the distance of the query to the cell that is possible to be accessed next is used as the key to maintain the priority queue. The exact lower bound requires the computation between a point and a hyperpolygon, which is generally time-consuming. As pointed in [2], the computation can be much more efficient if the projection directions along each path from the root to a leaf node are parallel or orthogonal, which we call the *orthogonality condition*. To make an orthogonal TP Tree, i.e., a TP tree satisfying orthogonality condition, we need to modify the enumeration scheme by checking if the candidate direction is orthogonal to or parallel with all the projection directions of its ancestors, which results in an increase of the time cost by a $\log n$ factor. In our experiments, we find that ANN search is still very good without any performance loss even if the orthogonality condition does not hold. Therefore, we simplify the implementation by intentionally ignoring the orthogonality condition and compute an approximate lower bound by directly accumulating the distance to the partition hyperplane with the distance lower bound of its parent node.

5.4 Adaptation to Low Dimensional Manifold

Space partitioning is one of key factors that affect the order of accessing the data points and determine the search efficiency. It is shown in [49] that PCA tree can reduce the diameter of the cell in a certain ratio given the low covariance dimension assumption, while KD tree cannot adapt to low dimensional manifolds. With regard to the maximum variance criterion, the principal direction in PCA tree is the best. The solution space of TP tree is apparently much larger than that of KD tree, and the principal trinary-projection direction is an approximation of the principal direction, which is not worse than the approximation using any coordinate axis. Although it remains unclear if TP tree adapts to low dimensional manifolds, our experiments show that our approach can produce good space partitioning that leads to better order to access data points.

5.5 Approximate and Exact NN Search

The proposed principal and randomized TP trees can also be applied to answer exact NN queries if the orthogonality condition holds. The priority search can be terminated when the minimum distance lower bound in the priority queue is larger than the best distance found currently, and thus the best NN found so far is the exact NN. A $(1+\varepsilon)$-approximate NN can be also found when the minimum distance in the priority queue is larger than $(1+\varepsilon)d_{\text{min}}$, where d_{min} is the best distance found so far. We have a theorem for $(1+\epsilon)$-approximate NN search over an orthogonal TP tree given as follows. The proof can be found from the supplemental material.

Theorem 3. *Consider a set $\mathcal{X}$ of data points in $\mathbb{R}^d$ indexed by an orthogonal trinary projection tree. Given a constant $\epsilon > 0$, there is a constant $c_{d,\varepsilon} \leqslant \lceil 1 + \frac{2\alpha\sqrt{d}}{\varepsilon}\rceil^d$, where α is the largest of aspect ratio of any cell, such that a $(1+\varepsilon)$-approximate nearest neighbor of a query $\mathbf{q}$ can be reported in $O(dc_{d,\varepsilon}\log n)$ time, and a sequence of k $(1+\varepsilon)$-approximate nearest neighbors of a query $\mathbf{q}$ can be computed in $O(d(c_{d,\varepsilon}+k)\log n)$ time.*

The lower bound of the distance between the query and the cell may not be tight enough, and hence the performance of exact NN and $(1+\varepsilon)$-approximate NN search is not satisfactory. As a result, ANN search within a time budget is practically conducted, equivalently terminating the search when a fixed number of data points have been accessed.

6 Experimental Results

6.1 Data Sets

Caltech. The Caltech 101 data set [15] contains about 9000 images and has been widely used for image classification. We extract the maximally stable extremal regions (MSERs) [32] for each image, and compute a 128-dimensional SIFT feature [30] for each MSER. On average, there are about 400 SIFT features for each image. In this way, we get a data set containing around $4000K$ SIFT feature points. In our experiment, we randomly sample $1000K$ points to build the reference data set. To formulate the query data set, we randomly sample $100K$ points from the original data points and guarantee that these query points do not appear in the reference data set.

Ukbench. The recognition benchmark images [36] consist of 2550 groups of 4 images each, most of which are about CD covers, indoor images and similar or identical objects, taken at different views. The images are all of size 640×480. We also extract MSERs and represent each MSER with a 128-dimensional SIFT feature. We randomly sample $1000K$ SIFT

TABLE 2
The description of the data sets used in our experiments.

	Caltech	Ukbench	Notre Dame	Oxford	Tiny images	PCA tiny images
dimension	128	128	128	128	384	64
#(reference points)	$1000K$	$1000K$	$400K$	$10M$	$1000K$	$1000K$
#(query points)	$100K$	$100K$	$60K$	$100K$	$100K$	$100K$

features for the reference data set, and $100K$ SIFT features as queries.

Notre Dame. The patch data set [21], associated with the Photo Tourism project [44], consists of local image patches of Flickr photos of various landmarks. The goal is to compute correspondences between local features across multiple images, which can then be provided to a structure-from-motion algorithm to generate 3D reconstructions of the photographed landmark [44]. Thus, one critical sub-task is to take an input patch and retrieve its corresponding patches within any other images in the database, which is essentially a large-scale similarity search problem. We use $400K$ image patches (represented by a 128-dimensional SIFT feature) from the Notre Dame Cathedral as the reference data set and $60K$ image patches as queries.

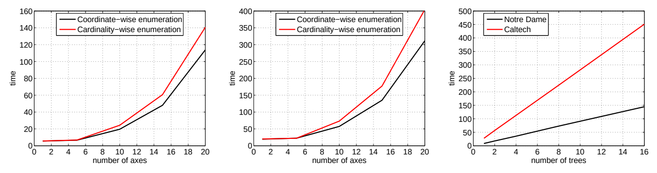
Oxford. The Oxford $5K$ data set [38] consists of 5062 high resolution images of 11 Oxford landmarks. There are about $16M$ SIFT features extracted from those images. We randomly sample $10M$ features as the reference data set and other $100K$ features as queries.

Tiny images. The tiny images data set consists of 80 million images, introduced in [46]. The sizes of all the images in this database are 32×32. Similar to [26], we use a global GIST descriptor [37] to represent each image, which is a 384-dimensional vector describing the texture within localized grid cells. We randomly sample $1000K$ images to build the reference data set and other $100K$ as queries from the remaining images. We also generate a data set, PCA tiny images, which is produced by reducing the dimension of the GIST feature to 64 using PCA.

The description of the data sets is summarized in Table 2. All the features are byte-valued except that the features in PCA tiny images are int-valued.

6.2 Evaluation Metric

We use the precision score to evaluate the search quality. For k-ANN search, the precision is computed as the ratio of the number of retrieved points which are contained in the true k nearest neighbors to k. The true nearest neighbors are computed by comparing each query with all the data points in the reference data set. We compare different algorithms by calculating the search precisions given the same search time, where the search time is recorded by varying the number of accessed data points. We report the performance in terms of search time vs. search precision. All the results are obtained with 64 bit programs on a $3.4G$ Hz quad core Intel PC with $16G$ memory.



(a) (b) (c)

Fig. 2. Construction cost. (a) and (b) show the comparison for construction cost (seconds) vs. different numbers of axes using coordinate-wise enumeration and cardinality-wise enumeration over Notre Dame and Caltech. (c) Construction cost vs. different numbers of trees for coordinate-wise enumeration.

6.3 Empirical Analysis

We present empirical results to show how various factors in our approach affect search precision and search efficiency.

6.3.1 *Construction Cost*

Figure 2 reports the time cost of constructing trinary projection trees when varying the number of used axes and the number of trees. From Figures 2(a) and 2(b) (one TP tree is built), one can see that the construction cost using more axes becomes larger, which conforms to the complexity analysis. In terms of the construction cost, the coordinate-wise enumeration scheme proposed in this paper is better than the cardinality-wise enumeration scheme that was used in [23]. From Figure 2(c) (5 axes are used), we can observe that the time taken by the coordinate-wise enumeration scheme increases linearly with respect to the number of trees.

6.3.2 *Coordinate-Wise Enumeration vs. Cardinality-Wise Enumeration*

We conduct experiments to compare the search performance over TP trees constructed using different enumeration schemes with the same parameters, $\bar{d} = 15$ leading axes used, $g = 15$ trinary-projection directions kept in each enumeration iteration, and 1 NN searched. From the comparison shown in Figure 3, we can see that the coordinate-wise enumeration scheme performs the best in terms of both search efficiency and precision, which is consistent to the previous analysis. Random enumeration means randomly sampling the trinary-projection direction $\mathbf{w}$ and the partition value b. Considering the less construction time using coordinate-wise enumeration, we choose it in the implementation.

6.3.3 *Dominant Axes*

We present the results to show how the number of axes used to form the projection direction for each internal node affects

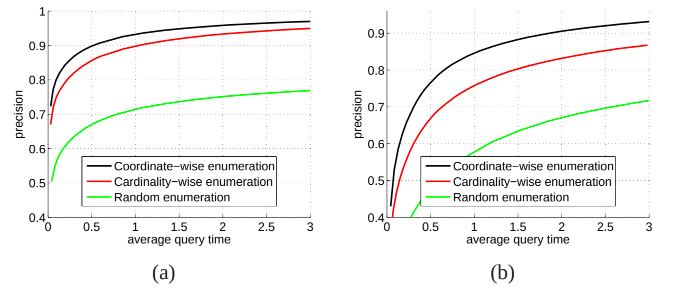


Fig. 3. Illustrating search performance using coordinate-wise enumeration and cardinality-wise enumeration in terms of precision vs. average query time (milliseconds) over (a) Notre Dame and (b) Caltech.

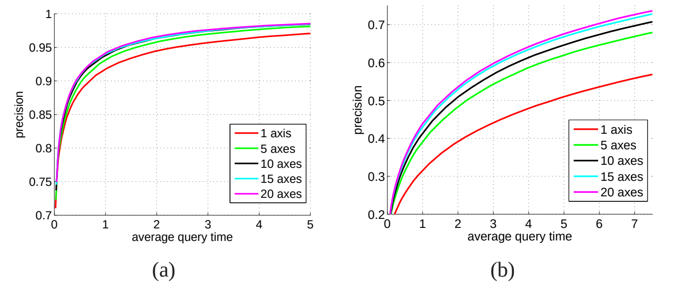


Fig. 4. Illustrating the performance of using different numbers of axes over (a) Notre Dame and (b) tiny images.

the search performance. The comparisons, obtained when one TP tree is used, are presented in Figure 4. There are several observations. Using more axes boosts the performance, and the improvement is even more significant especially when the data is in high dimensions as shown in Figure 4(b). The performance improvement becomes less significant when the number of used axes becomes larger.

6.3.4 *Multiple Randomized Trees*

This part presents experimental results to show using multiple randomized trees can lead to significant performance improvements. Figure 5 illustrates the comparisons over 1, 2, 4, 8, and 16 trees with 15 leading axes used for tree construction. As we can see, the performance with more trees is better. The precision improvement is quite significant when taking less query time. With more query time, the precision improvement becomes less significant. We can also see that the performances of 8 trees and 16 trees are very close.

6.4 Comparisons

We compare the search performance of our approach with state-of-the-art ANN search algorithms.

PCA tree. The PCA tree [45] is a binary spatial partition tree that chooses the principal direction as the projection direction at each internal node. It can yield compact space partitions. However, the projection operations at the internal nodes are very time-consuming, as it requires an inner product operation that takes $O(d)$ time. Consequently, the search performance is deteriorated. The priority search is used as a speedup trick in the implementation.

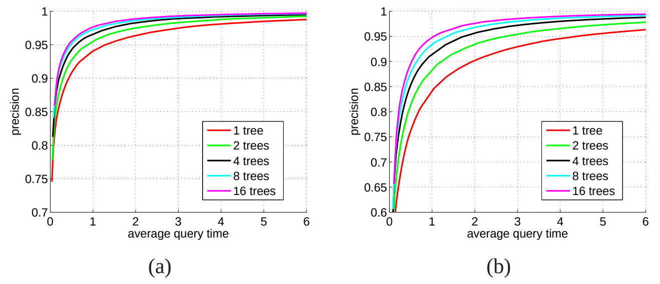


Fig. 5. Illustrating search performance when using different numbers of trees over (a) Notre Dame and (b) Caltech.

Vantage point tree. A vantage point (VP) tree [56] is a binary spatial partition tree that at each internal node segregates data points by choosing a position (the vantage point) in the space and dividing the data points into two partitions: those that are nearer to the vantage point than a threshold, and those that are not. The priority search is also used as a speedup trick in the implementation.

Spill tree. The spill tree [28] is a type of random projection tree. It generates the projection direction randomly. A key point of spill tree is that it allows overlapping partitions around the separating hyperplane. We implement the algorithm by following the description in [28].

Box-decomposition tree. This box-decomposition tree, shorted as BD tree [3], modifies the KD tree mainly in that, in addition to the splitting operation, there is a more general decomposition operation called shrinking for space partitioning. More details can be found from [3]. We report the experimental results by running their public implementation with a slight modification making the search proceed till a fixed number of points are accessed.

FLANN. FLANN [34] is a combination of multiple randomized KD trees and hierarchical k-means trees. It seeks the best configuration between them. We assume that its performance is better than the performances of KD trees and hierarchical k-means trees, thus we only report the results from FLANN.

Hashing. We also compare the performance with the hashing methods, E2LSH [11], multi-probe LSH [31], LSH forest [4], and spectral hashing [53]. The key idea of LSH is to hash the points using several hash functions to ensure that for each function the probability of collisions is much higher for objects that are close to each other than for those that are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point. It has been shown in [34] that randomized KD trees can outperform the LSH algorithm by about an order of magnitude. Multi-probe (MP) LSH is built on the LSH technique, but it intelligently probes multiple buckets that are likely to contain query results in a hash table. LSH forest represents each hash table by a prefix tree so that the number of hash functions per table can be adapted for different approximation distances. Spectral hashing aims to learn the hash functions according to the data distribution to build an effective hash table. As hashing methods are slower than our approach by about an order of magnitude, we report the comparisons with hashing

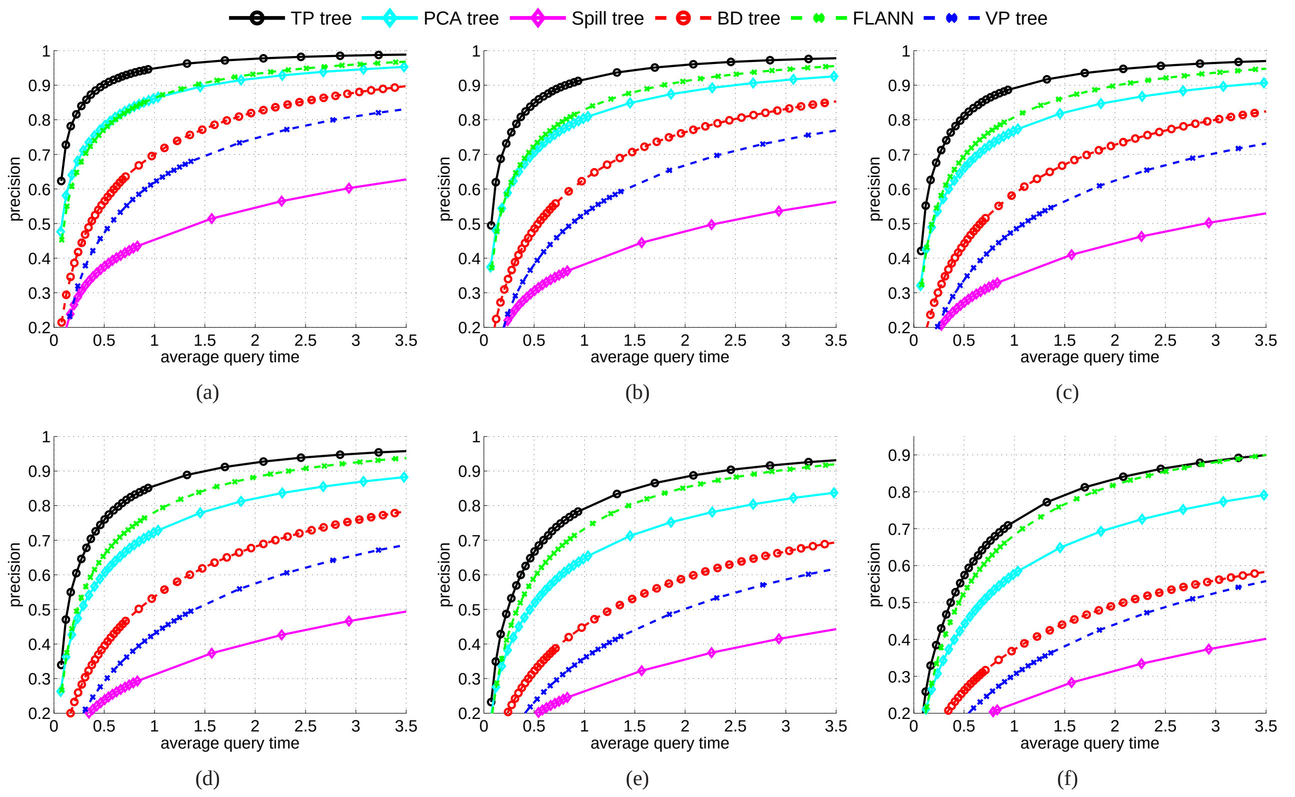


Fig. 6. Performance comparison over $1000K$ 128-dimensional features from Caltech. (a) 1-NN, (b) 5-NN, (c) 10-NN, (d) 20-NN, (e) 50-NN and (f) 100-NN.

based methods separately for clarity.

We first report the results of searching 1-NN, 5-NN, 10-NN, 20-NN, 50-NN and 100-NN on three data sets: $1000K$ 128-dimensional SIFT features over Caltech and Ukbench, and $1000K$ 384-dimensional GIST features over Tiny images. Searching for a small number of NNs is useful for patch matching and for more NNs is useful for similar image search. Our approach builds the tree using the mean as the partition value due to the cheap construction cost, uses 15 dominant axes and 10 random trees. The results of other approaches are obtained by using the well-tuned or auto-configured (if applicable) parameters. The comparisons are shown in Figures 6, 7 and 8, respectively. The horizontal axis corresponds to average query time (milliseconds), and the vertical axis corresponds to search precision.

From the results with 128-dimensional features as shown in Figures 6 and 7, our approach outperforms other approaches. Particularly, in the case of short query time, the superiority of our approach is much more significant, which is a desired property in the real search problems. The comparisons over $1000K$ high-dimensional GIST features are shown in Figure 8. The search for 384-dimensional features is actually more challenging. It can be seen that the improvement of our approach over other approaches is much more significant than for low-dimensional SIFT features. The precision of our approach is consistently higher than other methods at least 10% except PCA tree, which is a very significant improvement. One can see that the superiority of our approach, for searching different numbers of nearest neighbors, is consistent in the cases of both low and high dimensional cases. In contrast, other approaches cannot consistently produce satisfactory results.

We also conduct the experiments over a larger scale data set, $10M$ SIFT features over Oxford, shown in Figure 9. In this case, due to very high construction cost and much memory cost for PCA tree and Spill tree, we only report other three approaches. It can be seen that our approach consistently gets superior search performance.

Besides, we conduct the experiments to illustrate how preprocessing through PCA dimension reduction affects the search performance. We first do the PCA dimension reduction for the reference data set (tiny images) over which principal directions are computed, and get 64-dimensional features, forming a data set (PCA tiny images). We construct the index structure over 64-dimensional features. In the query stage, each query is also reduced by PCA to a 64-dimensional feature. The distances to the data points in the leaf nodes are evaluated over the original features and the ground truth is also computed over the original features. As Spill tree and VP tree perform very poorly, we only report the results of other three approaches. From the result shown in Figure 10, one can observe that our approach performs the best. Compared with the result without dimension reduction shown in Figure 8, the performances of all the approaches get improved. In comparison, the improvement of PCA tree is relatively small. This is as expected because PCA tree already has selected the principal directions for space partition while our approach as well as BD tree benefit a lot from the better coordinate axes produced by PCA dimension reduction.

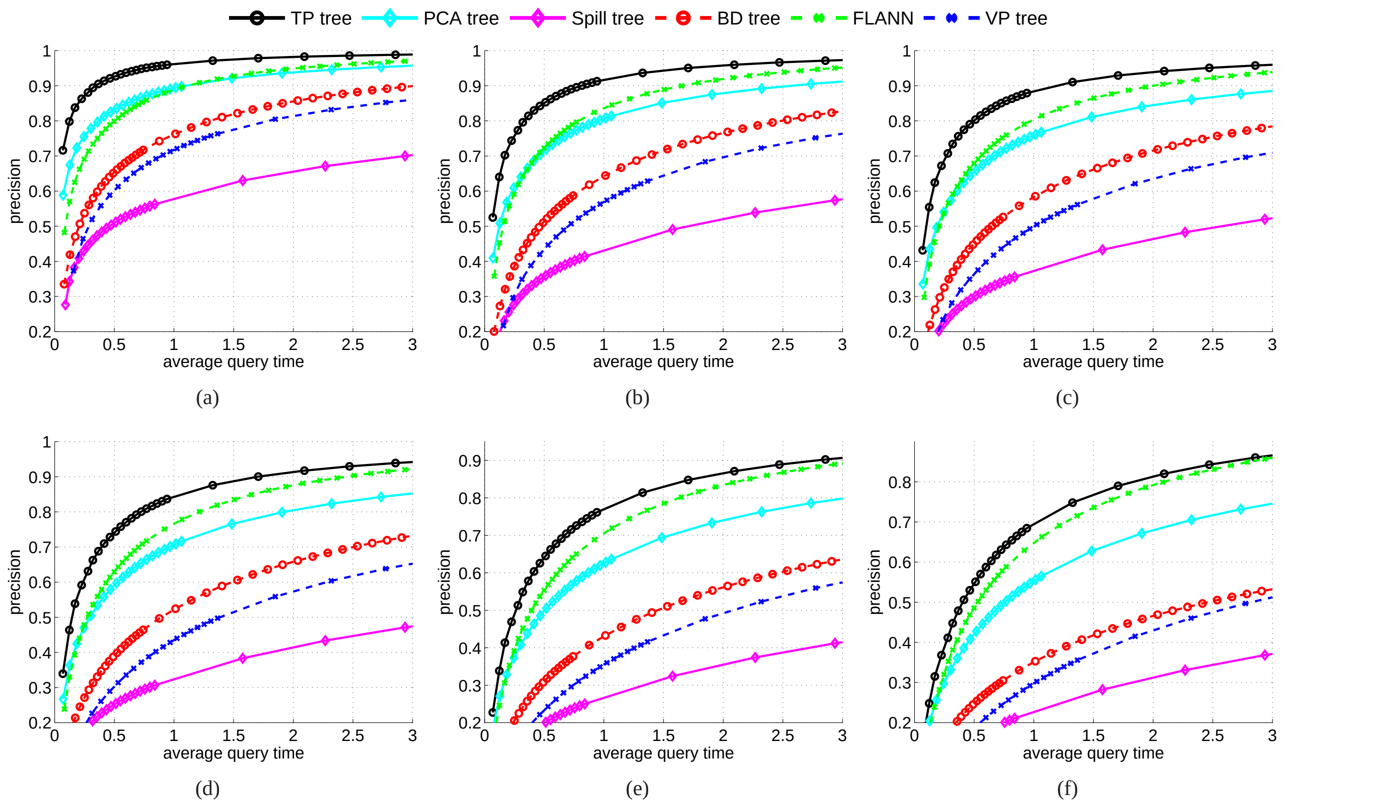


Fig. 7. Performance comparison over $1000K$ 128-dimensional features from Ukbench. (a) 1-NN, (b) 5-NN, (c) 10-NN, (d) 20-NN, (e) 50-NN and (f) 100-NN.

The above experimental results indicate that (1) our approach achieves a large improvement in the case of searching a small number of nearest neighbors and (2) the improvement is relatively small in the case of searching a large number of nearest neighbors. The first point implies that our approach is powerful to discriminate the points that are near to the query. The second point means that most approaches are able to discriminate the near neighbors from the far neighbors.

Last, we report the comparison with hashing methods. As hashing methods are very slow, we report the performance with the time axis in a logarithmic scale, in order to make the comparisons clear. We include the results searching for 1-NN and 20-NN, over two data sets: Caltech with 128-dimensional features and tiny images with 384-dimensional features as we observed that the conclusions for other k-NNs and other data sets remain valid. The comparisons are shown in Figure 11. It can been observed that hashing methods perform poorly, and are much slower (even several orders) than our approach. MP LSH and SH perform the second best, which is reasonable because MP LSH performs the best-first search scheme and SH learns better space partitioning than other hash algorithms. The superiority of our approach comes from good space partitioning and the best-first search scheme.

7 Conclusion

In this paper, we present a novel hierarchical spatial partition tree for approximate nearest neighbor search. The key idea is using a trinary projection direction, a linear combination of a few coordinate axes with weights being -1 or 1, to form the partition hyperplane. The superiority of our approach comes from two aspects: (1) fast projection operation at internal nodes in traversing, only requiring a few addition/subtraction operations, which leads to high search efficiency, and (2) good space partition guaranteed by a large variance along the projection direction for partitioning data points, which results in high search accuracy. The data sets used in our experiments and the implementation of our approach are publicly available from the project page http://research.microsoft.com/~jingdw/SimilarImageSearch/tptree.html.

Acknowledgements

Jian Li is supported in part by the National Basic Research Program of China Grant 2011CBA00300, 2011CBA00301, the National Natural Science Foundation of China Grant 61202009, 61033001, 61061130540, 61073174.

References

[1] D. Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *J. Comput. Syst. Sci.*, 66(4):671–687, 2003.
[2] S. Arya and D. M. Mount. Algorithms for fast vector quantizaton. In *Data Compression Conference*, pages 381–390, 1993.
[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. *J. ACM*, 45(6):891–923, 1998.
[4] M. Bawa, T. Condie, and P. Ganesan. LSH forest: Self-tuning indexes for similarity search. In *WWW*, pages 651–660, 2005.

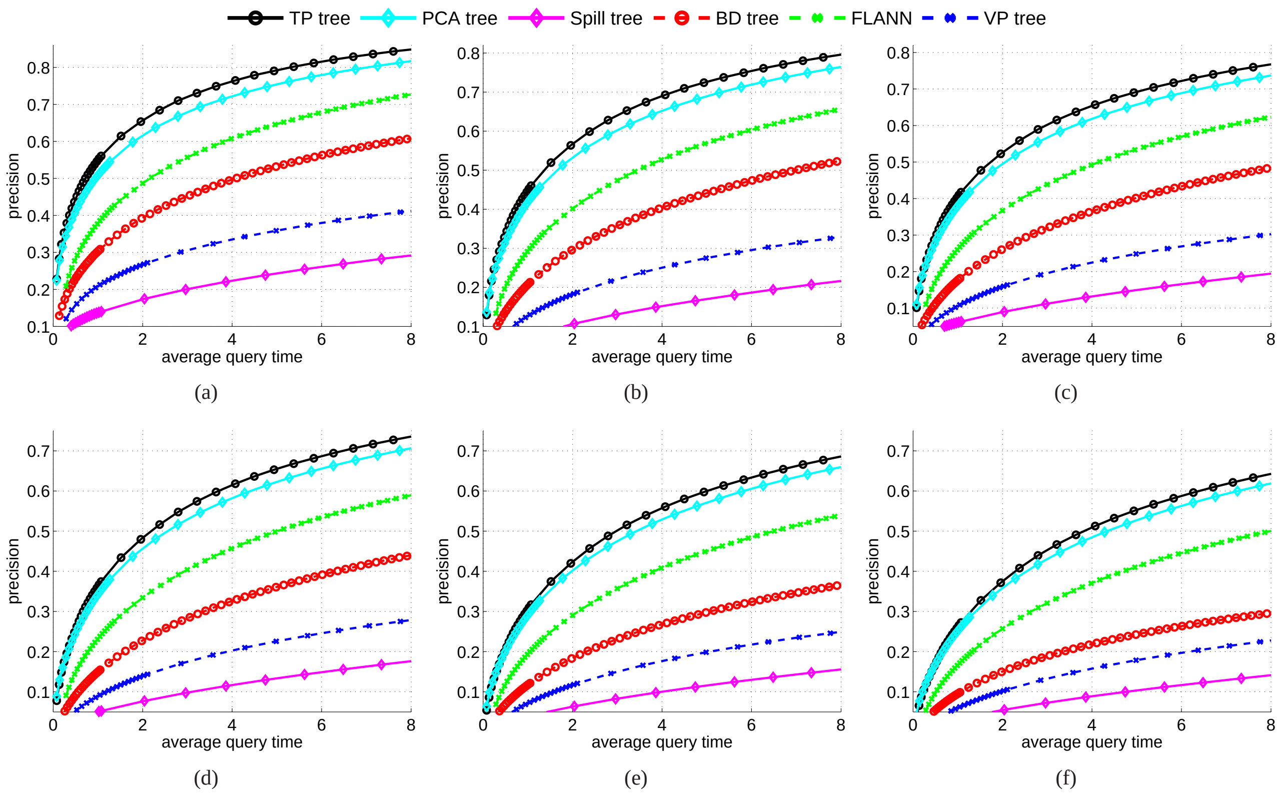


Fig. 8. Performance comparison over $1000K$ 384-dimensional features from tiny images. (a) 1-NN, (b) 5-NN, (c) 10-NN, (d) 20-NN, (e) 50-NN and (f) 100-NN.

[5] J. S. Beis and D. G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, pages 1000–1006, 1997.
[6] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.
[7] J. Besag. On the statistical analysis of dirty pictures. *J. Roy. Stat. Soc*, 48(3):259–302, 1986.
[8] M. Brown and D. G. Lowe. Recognising panoramas. In *ICCV*, pages 1218–1227, 2003.
[9] L. Cayton and S. Dasgupta. A learning framework for nearest neighbor search. In *NIPS*, 2007.
[10] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *STOC*, pages 537–546, 2008.
[11] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Symposium on Computational Geometry*, pages 253–262, 2004.
[12] M. de Berg, T. Eindhoven, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.
[13] A. A. Efros and W. T. Freeman. Image quilting for texture synthesis and transfer. In *SIGGRAPH*, pages 341–346, 2001.
[14] C. Faloutsos and K.-I. Lin. FastMap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *SIGMOD Conference*, pages 163–174, 1995.
[15] L. Fei-Fei, R. Fergus, and P. Perona. Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories. In *CVPR 2004 Workshop on Generative-Model Based Vision*, 2004.
[16] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.
[17] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.
[18] A. Frome, Y. Singer, F. Sha, and J. Malik. Learning globally-consistent local distance functions for shape-based image retrieval and classification. In *ICCV*, pages 1–8, 2007.
[19] J. Hays and A. A. Efros. Scene completion using millions of photographs. *ACM Trans. Graph.*, 26(3):4, 2007.
[20] J. He, W. Liu, and S.-F. Chang. Scalable similarity search with optimized kernel hashing. In *KDD*, pages 1129–1138, 2010.
[21] G. Hua, M. Brown, and S. A. J. Winder. Discriminant embedding for local image descriptors. In *ICCV*, pages 1–8, 2007.
[22] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *CVPR*, 2008.
[23] Y. Jia, J. Wang, G. Zeng, H. Zha, and X.-S. Hua. Optimizing kd-trees for scalable visual descriptor indexing. In *CVPR*, pages 3392–3399, 2010.
[24] W. Johnson and J. Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. *Contemporary Mathematics*, 26:189–206, 1984.
[25] B. Kulis and T. Darrells. Learning to hash with binary reconstructive embeddings. In *NIPS*, pages 577–584, 2009.
[26] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, pages 2130–2137, 2009.
[27] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, and H.-Y. Shum. Real-time texture synthesis by patch-based sampling. *ACM Trans. Graph.*, 20(3):127–150, 2001.
[28] T. Liu, A. W. Moore, and A. G. Gray. New algorithms for efficient high-dimensional nonparametric classification. *Journal of Machine Learning Research*, 7:1135–1158, 2006.
[29] T. Liu, A. W. Moore, A. G. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004.
[30] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.
[31] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.
[32] J. Matas, O. Chum, M. Urban, and T. Pajdla. Robust wide baseline stereo from maximally stable extremal regions. In *BMVC*, 2002.
[33] A. W. Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *UAI*, pages 397–405, 2000.
[34] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISSAPP (1)*, pages 331–340, 2009.
[35] G. Navarro. Searching in metric spaces by spatial approximation. *VLDB J.*, 11(1):28–46, 2002.
[36] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *CVPR (2)*, pages 2161–2168, 2006.
[37] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145–175, 2001.

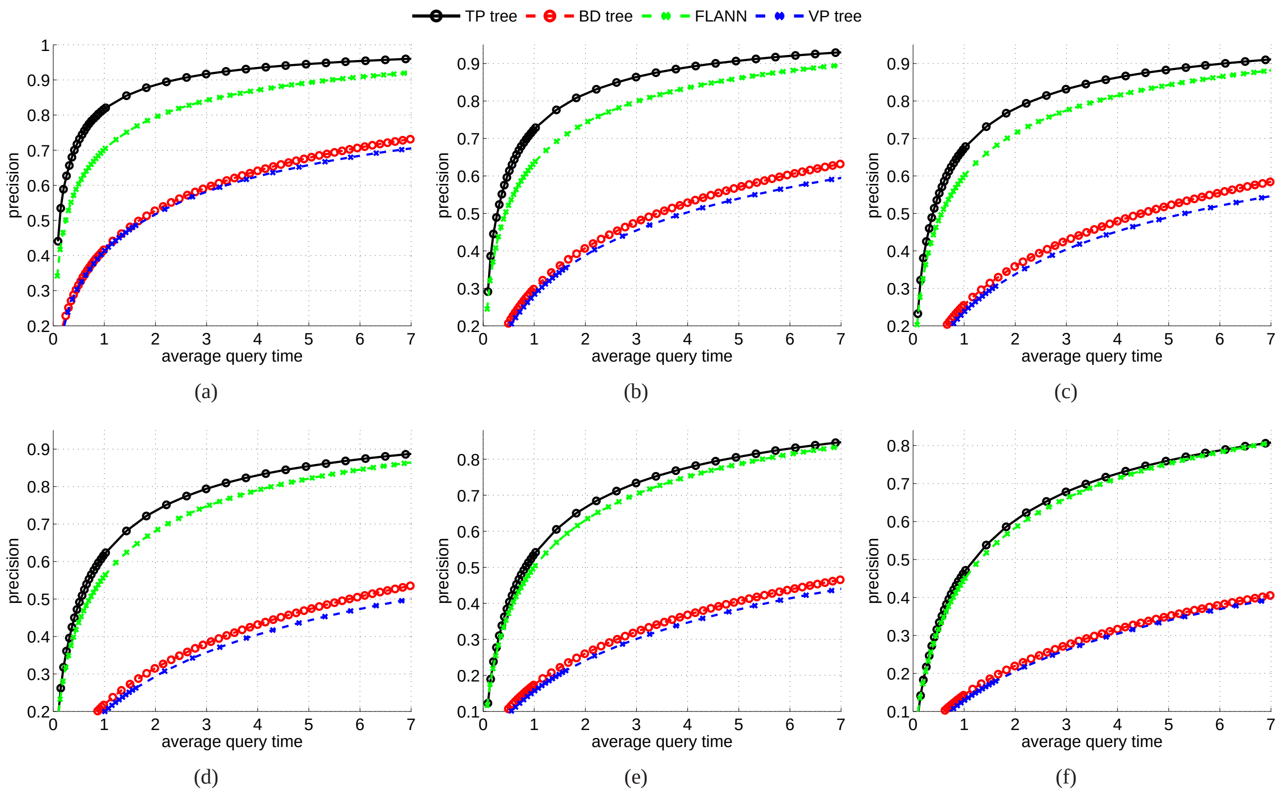
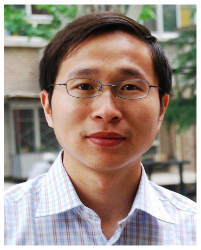
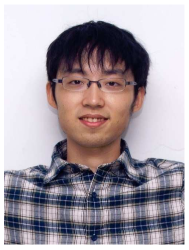


Fig. 9. Performance comparison over $10M$ 128-dimensional features from Oxford. (a) 1-NN, (b) 5-NN, (c) 10-NN, (d) 20-NN, (e) 50-NN and (f) 100-NN.

[38] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *CVPR*, 2007.
[39] M. Raginsky and S. Lazebnik. Locality sensitive binary codes from shift-invariant kernels. In *NIPS*, 2009.
[40] H. Samet. *Foundations of multidimensional and metric data structures*. Elsevier, Amsterdam, 2006.
[41] T. B. Sebastian and B. B. Kimia. Metric-based shape retrieval in large databases. In *ICPR (3)*, pages 291–296, 2002.
[42] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*. The MIT press, 2006.
[43] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *CVPR*, 2008.
[44] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: Exploring photo collections in 3D. *ACM Trans. Graph.*, 25(3):835–846, 2006.
[45] R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.
[46] A. B. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(11):1958–1970, 2008.
[47] G. T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.
[48] W. Tu, R. Pan, and J. Wang. Similar image search with a tiny bag-of-delegates representation. In *ACM Multimedia*, pages 885–888, 2012.
[49] N. Verma, S. Kpotufe, and S. Dasgupta. Which spatial partition trees are adaptive to intrinsic dimension? In *UAI*, pages 565–574, 2009.
[50] J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for scalable image retrieval. In *CVPR*.
[51] J. Wang and S. Li. Query-driven iterated neighborhood graph search for large scale indexing. In *ACM Multimedia*, pages 179–188, 2012.
[52] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li. Scalable k-nn graph construction for visual descriptors. In *CVPR*, pages 1106–1113, 2012.
[53] Y. Weiss, A. B. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2008.
[54] H. Xu, J. Wang, Z. Li, G. Zeng, S. Li, and N. Yu. Complementary hashing for approximate nearest neighbor search. In *ICCV*, pages 1631–1638, 2011.
[55] K. Yamaguchi, T. L. Kunii, and K. Fujimura. Octree-related data structures and algorithms. *IEEE Computer Graphics and Applications*, 4(1):53–59, 1984.
[56] P. N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *SODA*, pages 311–321, 1993.
[57] H. Zhang, A. C. Berg, M. Maire, and J. Malik. SVM-KNN: Discriminative nearest neighbor classification for visual category recognition. In *CVPR (2)*, pages 2126–2136, 2006.



Jingdong Wang received the BSc and MSc degrees in Automation from Tsinghua University, Beijing, China, in 2001 and 2004, respectively, and the PhD degree in Computer Science from the Hong Kong University of Science and Technology, Hong Kong, in 2007. He is currently a researcher at the Media Computing Group, Microsoft Research Asia. His areas of interest include computer vision, machine learning, and multimedia search. At present, he is mainly working on the Big Media project, including large-scale indexing and clustering, and Web image search and mining. He is an editorial board member of Multimedia Tools and Applications.



Naiyan Wang received the BS degree in computer science from Zhejiang University, China, in 2011. He is now pursuing his PHD degree in the Hong Kong University of Science and Technology. His research interests include sparse representations, matrix factorizations, manifold learning and their applications in computer vision and data mining.

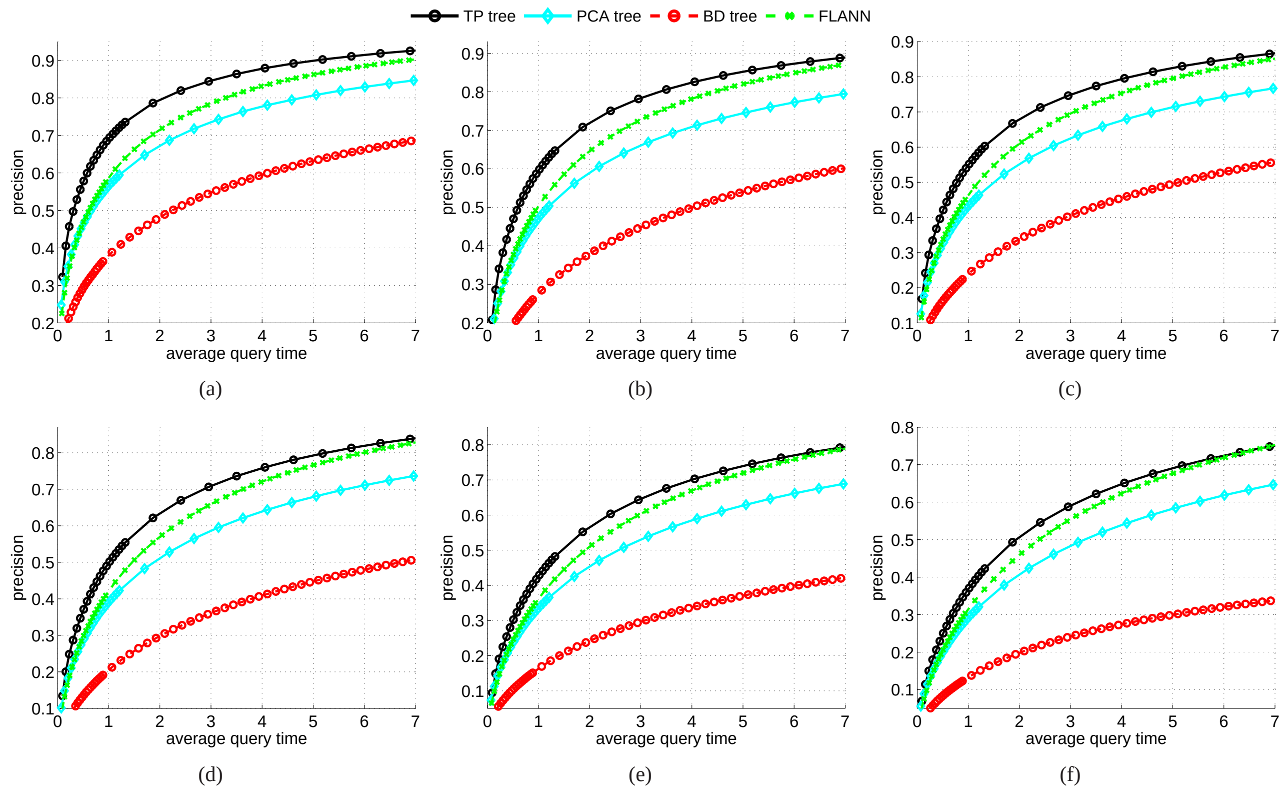


Fig. 10. Performance comparison over PCA tiny images. (a) 1-NN, (b) 5-NN, (c) 10-NN, (d) 20-NN, (e) 50-NN and (f) 100-NN.

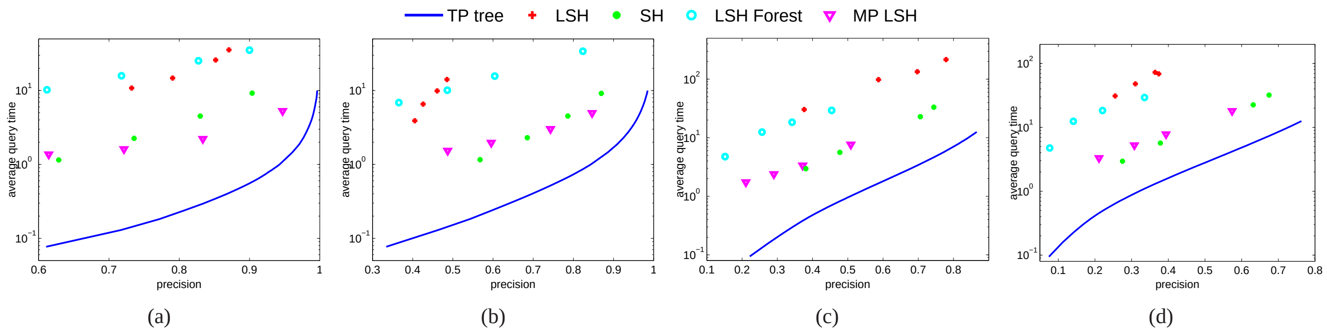
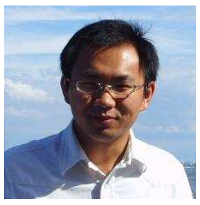


Fig. 11. Performance comparison with various hashing methods. (a) and (b) correspond to the comparison of 1-NN and 20-NN over Caltech, and (c) and (d) correspond to the comparison of 1-NN and 20-NN over tiny images.

You Jia received the BS degree in computer science from Peking University, China, in 2010, and the MS degree in robotics from Carnegie Mellon University in 2011. Upon graduation, he joined Facebook as a software engineer. During his brief research career, he was focused on the problems of large scale image retrieval and later text detection in the wild.



Jian Li is an Assistant Professor at Institute for Interdisciplinary Information Sciences, Tsinghua University. He got his BSc degree from Sun Yat-sen (Zhongshan) University, China, MSc degree in computer science from Fudan University, China and PhD degree in the University of Maryland, USA. His research interests lie in the areas of algorithms, databases and wireless sensor networks. He co-authored several research papers that have been published in major computer science conferences and journals. He received the best paper awards at VLDB 2009 and ESA 2010.

Gang Zeng obtained his B.S. degree from School of Mathematical Sciences at Peking University, and his Ph.D. degree from Department of Computer Science and Engineering at the Hong Kong University of Science and Technology in 2001 and 2006 respectively. He worked as a postdoc research fellow in the BIWI Computer Vision Laboratory at ETH Zurich from 2006 to 2008. He is currently working as a research professor at the Key Laboratory on Machine Perception at Peking University. He is interested in computer vision and graphics, specifically in three-dimensional scene reconstruction, image-based or interactive modeling and rendering, and unsupervised image segmentation. He served as the program chair of the 7th Joint Workshop on Machine Perception and Robotics 2011, and the organizing committee member of the 9th Asian Conference on Computer Vision 2009, the Workshop on Computer Vision 2009, and the International Symposium on Machine Perception and Cognition 2010. He is also the assistant director of the MOE-Microsoft Key Laboratory of Statistics and Information Technology of Peking University.

Hongbin Zha received BE degree in electrical engineering from Hefei University of Technology, China, in 1983 and MS and PhD degrees in electrical engineering from Kyushu University, Japan, in 1987 and 1990, respectively. After working as a research associate at Kyushu Institute of Technology, he joined Kyushu University in 1991 as an associate professor. He was also a visiting professor in Centre for Vision, Speech, and Signal Processing, Surrey University, United Kingdom, in 1999. Since 2000, he has been a professor at Key Laboratory of Machine Perception (MOE), Peking University, China. His research interests include computer vision, digital geometry processing, and robotics. He has published more than 250 technical publications in journals, books, and international conference proceedings. He received the Franklin V. Taylor Award from IEEE Systems, Man, and Cybernetics Society in 1999.

Xian-Sheng Hua became a Principal Research and Development Lead in Multimedia Search for the Microsoft search engine, Bing, in 2011. He leads a team that designs and delivers leading-edge media understanding, indexing and searching features. He joined Microsoft Research Asia in 2001 as a researcher. Since then, his research interests have been in the areas of multimedia search, advertising, understanding, and mining, as well as pattern recognition and machine learning. He has authored or co-authored more than 200 research papers in these areas and has filed more than 60 patents. He received his BS in 1996 and PhD in applied mathematics in 2001 from Peking University, Beijing. He served as an associate editor of IEEE Transactions on Multimedia in 2007-2011, and now is serving as an associate editor of ACM Transactions on Intelligent Systems and Technology, an editorial board member of Advances in Multimedia and Multimedia Tools and Applications, and an editor of Scholarpedia (multimedia category). He was vice program chair; workshop organizer; senior TPC member and area chair; and demonstration, tutorial, and special session chairs and PC member of many more international conferences. He serves as a program co-chair for ACM Multimedia 2012 and IEEE ICME 2012.